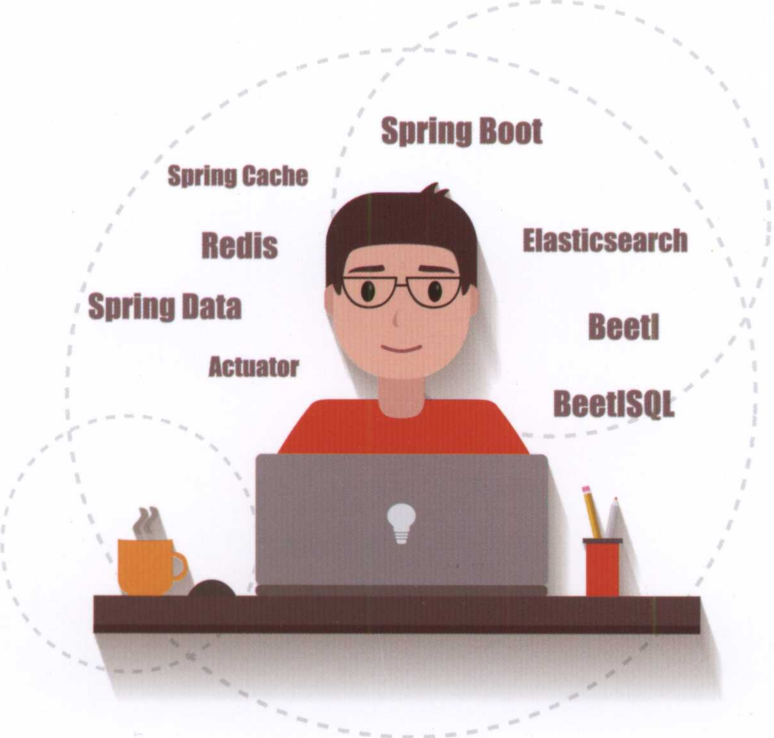
Broadview®
www.broadview.com.cn




# Spring Boot 2 精髓

## 从构建小系统到架构分布式大系统

李家智 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.COM

# Spring Boot 2 精髓

## 从构建小系统到架构分布式大系统

李家智 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

## 内 容 简 介

Spring Boot 是目前 Spring 技术体系中炙手可热的框架之一，既可用于构建业务复杂的企业应用系统，也可以开发高性能和高吞吐量的互联网应用。Spring Boot 框架降低了 Spring 技术体系的使用门槛，简化了 Spring 应用的搭建和开发过程，提供了流行的第三方开源技术的自动集成。

本书系统介绍了 Spring Boot 2 的主要技术，侧重于两个方面，一方面是极速开发一个 Web 应用系统，详细介绍 Spring Boot 框架、Spring MVC、视图技术、数据库访问技术，并且介绍多环境部署、自动装配、单元测试等高级特性；另一方面，当系统模块增加，性能和吞吐量要求增加时，如何平滑地用 Spring Boot 实现分布式架构，也会在本书后半部分介绍，包括使用 Spring 实现 RESTful 架构，在 Spring Boot 框架下使用 Redis、MongoDB、ZooKeeper、Elasticsearch 等流行技术，使用 Spring Session 实现系统水平扩展，使用 Spring Cache 提高系统性能。

阅读本书的人，可以是 Java 新手，从未使用过任何 Spring 技术的工程师。也可以是用过 Spring，但想进一步了解 Spring Boot 的开发者。如果你已经使用过 Spring Boot，那么本书也非常适合你全面深入了解 Spring Boot。

**未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。**
**版权所有，侵权必究。**

**图书在版编目（CIP）数据**

Spring Boot 2 精髓：从构建小系统到架构分布式大系统 / 李家智著. —北京：电子工业出版社，2017.10
ISBN 978-7-121-32825-1

Ⅰ. ①S… Ⅱ. ①李… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 243885 号

责任编辑：陈晓猛
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　　　邮编：100036
开　　本：787×980　1/16　　印张：24　　　　字数：460 千字
版　　次：2017 年 10 月第 1 版
印　　次：2017 年 10 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

# 专家评价

作者集其近二十年来沉浸于技术架构的理论探索和实践经验，特别是近年来扎根于互联网金融机构对于开源技术架构的前沿研究实践，方成此心血之作。全书由易及难、自浅入深，为读者徐徐展开基于 Spring Boot 2.0 构建企业复杂应用的恢弘篇章。此书非常适合作为开发人员及架构师从新手到高手、自低阶至高阶的重要指导书和参考书。

—— 东方资产信息科技部 贺锦

如何快速适应业务的变化发展一直是每个企业 IT 部门都面临的困扰，现在 Spring Boot 开发框架提供了最新的、经过实践验证的技术解决方案。

本书作者，一位近二十年 Java 程序员和架构师，结合他高超的技术能力和丰富的实战经验，给读者精心阐述了 Spring Boot 2.0 从初级快速构建系统到复杂的分布式系统开发的完整开发技术，本书不仅是开发人员不可多得的教科书，对非软件企业的 IT 人员也有极强的参考价值。

—— 东方资产信息科技部 黄友平

过去几年，微服务架构在软件开发领域逐渐深入人心，Spring Boot 在经历了快速演变之后，正在成为 Java 微服务开发的主流成熟框架。本书对 Spring Boot 的特性进行了全方位讲解，辅以大量翔实的案例，对分布式系统开发和应用提供了实战指导。书中还详细介绍了作者倾注了大量心血研发的开源软件 Beetl 和 BeetlSQL，它们易于与 Spring Boot 集成，并已被众多大公司采纳使用。本书对于开发人员和架构师来说，都极具参考价值。

——《Kubernetes 权威指南》作者/HPE 高级顾问 龚正

和家智相识多年，他是我所认识的非职业足球运动员中球商最高的，多年来他一直深耕于技术的第一线，有着丰富的技术储备，是我非常钦佩的老大哥。

我们曾经在同一家公司的同一个项目集效力，都非常喜欢踢球，都出了书，不得不说很神奇。

这本书的内容非常丰富，也是作者多年潜心钻研的积累，这本书和我的书有一些联系，将 Spring Boot 进行了展开讲解，既有广度，也有深度，非常值得技术人员去学习。

——《微服务那些事儿》作者　纪晓峰

Spring Boot 上手简单、功能丰富、易于扩展，可谓目前业界快速开发、快速生产的利器；然而，它的版本演进非常迅速，中文文档质量也参差不齐。本书由浅入深地讲解了 Spring Boot，帮助读者系统理解 Spring Boot。不仅如此，本书对 2.0 版本带来的新特性亦有非常详尽的描述，绝对值得一读。

——《Spring Cloud 与 Docker 微服务架构实战》作者　周立

和家智相识有五年了，最早是因为模板引擎技术结识。家智是国内顶级的模板引擎专家，也是我认识的为数不多的 Spring 技术专家，他在这两方面都曾带给我不同角度的思考与印证，让我获益匪浅。本书是家智二十年技术钻研的一次厚积薄发，其中不仅从作者自身的经验详细讲述了 Spring Boot 框架，还引入了作者在模板引擎、ORM 以及单元测试方面多年研发的开源作品，是 Java 程序员和架构师不可多得的参考资料。

—— ACTFramework 作者　罗格林

十多年前，Spring 颠覆了传统的 Java EE 技术，迎来了 Java 企业级应用开发的春天，然而今天的 Spring Boot 却站在 Spring 巨人的肩膀上，让我们可以更高效地开发与交付。李家智是著名开源框架 Beetl 的作者，他写的《Spring Boot 2 精髓：从构建小系统到架构分布式大系统》一定非常精彩。

—— 特赞科技 CTO　黄勇

最早熟知作者是从 Beetl 模板引擎开始的，当时正在寻找一个易用高效的模板引擎，Beetl 几乎满足了我所有的需要。同时也十分敬佩作者在开源项目上的认真与负责。本书可以说是作者多年的心血所著，从 Spring Boot 的前世今生到使用扩展，都做了非常全面而易懂的概括，细节上也秉承了作者的细致与认真，讲解清晰并语言干练，既适合初学者系统化学习，也适合有经验的工程师作为参考。

—— 开源工具集 Hutool 作者　路小磊

从事 Web 开发有些年头了，经过技术选型，Spring Boot 走入了我们的视野，开箱即用，非常方便，也是目前很多大公司的选择之一。除了研究源码，如果有一本关于 Spring Boot 的指导书籍，既可以方便地解决开发中的问题，又可以帮助读者掌握 Spring Boot，提高生产效率。

家智兄的这本书正是这样不可多得的优秀资源，本书是家智兄多年钻研的技术积累，书中详细讲解了 Web 开发的各个知识点，包含 Web 请求处理、ORM 处理、Redis 缓存、MongoDB、Elasticsearch、ZooKeeper、监控等方面的知识点。相信读者在仔细阅读并掌握本书的知识点后，可以极大地提高自身的 Web 开发水平，为读者的软件开发事业助一臂之力！

—— 上海秦苍（买单侠）基础架构组架构师　刘志强

作者在 Java EE 体系内的多年实战经验使得本书的内容极具价值，书中清晰细致地讲解了快速构建 Web 应用系统的各个知识点，尤其是在后端模板引擎和 ORM 两个章节中，作者以自己的两款成熟开源产品 Beetl 和 BeetlSQL 为切入点进行讲解，剖析角度十分新颖并且有启发性。

通过这本书可以学习到关于 Spring Boot 框架的核心技术，从而掌握快速构建分布式 Web 应用的必备知识。无论你是 Spring Boot 新手，还是已经使用过 Spring Boot 的开发者，相信都可以从这本书中受益。

—— XXL-JOB 系列作者　许雪里

近两年来，随着微服务的兴起，Spring Boot 突然流行起来了，越来越多的公司采用这一技术，其已经成为大多数 Java 微服务开发者的首选开源框架。Spring Boot 有非常显著的特点：配置简单，易于开发，可快速部署。本书结合丰富的实例，从 Spring Boot 的快速开发 Web 应用

入手，逐渐深入地分析 Spring Boot 的高级特性，最后再重点介绍分布式架构的应用。通过深入浅出的阐述，让你从单体应用到分布式、微服务都有全方位的了解，是不可多得的一本好书，当然我认为最重要的还是作者耗费心血的开源项目 Beetl 和 BeetlSQL。

——《分布式数据库架构及企业实践——基于 Mycat 中间件》作者，
开源中间件 Mycat 负责人　周继锋

Spring 风靡多年，Spring Boot 在最近几年微服务框架浪潮下更是出尽风头，本书作者由浅入深地把 Spring Boot 2.0 各种特性阐述得淋漓尽致，不管你是 Spring Boot 新手还是老司机都值得一读。Java Web 后端也好，App 后台也罢，甚至独立后台应用，等等，Spring Boot 都是你不可或缺的高效率工具。

移动易项目团队深深的体会就是使用了 Spring Boot 可以节省 50%以上的代码。

—— 上海亿琪软件有限公司 CEO，移动易开源项目负责人，
华为开发者社区专家（HDE）　褚建琪

# 前言

Java 的各种开发框架发展了很多年，影响了一代又一代的程序员，现在无论是程序员，还是架构师，使用这些开发框架都面临着两方面的挑战。

- 一方面是要快速开发出系统，这就要求使用的开发框架尽量简单，无论是新手还是老手都能快速上手，快速掌握页面渲染、数据库访问等常用技术。也要求开发框架能尽量多地集成第三方工具，以便信手拈来。最后，还希望在开发调试过程中，方便代码更改后能快速重启。
- 另外一方面，当系统模块增加，用户使用量增加时，面对这样的挑战，系统拆分成为新的架构，程序员和架构师当然不希望换掉已有的开发框架，希望能由小而美的系统过渡到大而强的分布式系统。

环顾当前 Java 开源世界中的流行技术框架，能同时胜任这项工作的微乎其微，Play 和 ActFramework 都是不错的选择，国内的 Nutz 和 JFinal 的口碑也不错。但能同时满足快速开发和分布式系统架构的框架，还是群众基础最好、功能最全、基于 Spring 技术的 Spring Boot 框架。

## 内容介绍

本书系统介绍了 Spring Boot 2.0 的主要技术，侧重于两个方面，一方面是极速开发一个 Web 应用系统（第 1～6 章，包含 Spring 介绍、MVC、视图技术、数据库访问技术），随后介绍了 Spring Boot 的高级特性（第 7～9 章），包括多环境部署、自动装配、单元测试等技术。另外一方面，当系统模块增加，性能和吞吐量要求增加时，如何平滑地用 Spring Boot 来实现分布式架构，会在本书的第 10～17 章介绍。

阅读本书的读者，可以是 Java 新手，从未使用过任何 Spring 技术。也可以是用过 Spring，但想进一步了解 Spring Boot 的开发者。如果你已经使用过 Spring Boot，那么本书也非常适合你全面深入了解 Spring Boot。

希望读者阅读完本书后，既能轻松快速构建 Web 应用系统，也能掌握分布式系统架构的实现。

**上半部分介绍 Spring Boot 的基础技术。**

**第 1 章：** 介绍 Java EE，然后指出其缺点，引入了流行的 Spring，同时也说明 Spring 经过这么多年发展后暴露的一些缺点，从而引出 Spring Boot，并以两个简要例子作为说明。

**第 2 章：** 对 Spring Boot 应用的开发环境做了说明，包括 Java 开发环境的安装和配置，Maven 的安装和配置，设置国内仓库镜像，还有常用的 Maven 命令。本章最后介绍 Spring 历史以及现有开发团队，并介绍 Spring 框架的 AOP 和 IoC 两个核心技术

**第 3 章：** 介绍 MVC 技术，前半部分重点介绍 URL 映射到 Controller，以及映射到 Controller 方法的参数、参数类型转化、参数验证。后半部分简单介绍 MVC 中的视图技术 Freemaker、Beetl，以及 Jackson 序列化技术。Beetl 和 Jackson 将在第 4 章详细介绍。

**第 4 章：** 介绍笔者的开源技术 Beetl 后端模板引擎，作为国内流行的模板引擎之一，具有简单易学、功能/性能强大、支持 MVC 分离开发等特点。另外一部分详细介绍 Jackson 的 JSON 序列化技术。Jackson 不仅作为 Spring MVC 中的 JSON 默认工具，也是 Spring Boot 分布式技术中常采用的 JSON 序列化技术。

**第 5 章：** 介绍以 SQL 为中心的数据库访问工具 BeetlSQL，这是笔者的另外一款流行 Dao 工具，SQL 在 markdown 文件中管理，内置增删改查、轻量级 ORM 功能、代码生成、主从支持、跨多种数据库等特点，适合那些更喜欢以 SQL 方式访问数据库的开发者。

**第 6 章：** 介绍以面向对象为中心的数据库访问工具 Spring Data JPA。本章由易到难，先从 Spring Data 提供的功能入手，介绍如何完成数据库简单的增删改查功能，然后引入 JPA 来解决应用中不可避免的复杂 SQL 查询。

**第 7 章：** 介绍 Spring Boot 高级特性，如常用的 Spring Boot 的配置、日志配置、应用配置的读取、Spring Boot 自动装配技术和 Spring Boot Starter 实现。

**第 8 章：** 介绍如何部署 Spring Boot 应用，包括可执行 jar，以及通过 war 部署到应用服务器上。应用经常面对多个环境，如开发、测试，还有准线上、线上，以及多个 Demo 环境，Spring Boot 提供 Profile 来实现多环境部署。

**第 9 章：** 介绍单元测试概念，以及 Spring Boot 下的单元测试支持，包括 MVC 单元测试、Mock 测试，以及面向数据库应用的测试方案。

**下半部分介绍与 Spring Boot 相关的分布式技术。**

**第 10 章：** 介绍 RESTful 风格的架构，然后介绍 Spring Boot 如何集成以提供 REST 服务，使用 RestTemplate 调用 REST 服务。本章最后也重点介绍了 Swagger 3.0 技术，以方便 REST 的接口的交流、开发和测试。

**第 11 章：** 介绍 MongoDB 的安装和使用，然后介绍 Spring Boot 如何集成 MongoDB，同时还介绍了如何用 MongoTemplate 访问 MongoDB。

**第 12 章：** 介绍 Redis 服务器的安装和使用，Redis 常用的数据结构和操作命令。然后介绍 Spring Boot 如何集成 Redis，如何使用 RedisTemplate 来操作 Redis。本章后半部分深入介绍了 RedisTemplate 提供的序列化机制。

**第 13 章：** 介绍 Elasticsearch 的安装和使用，Elasticsearch 既具有全文搜索功能，也能像 MongoDB 那样，具备 NoSQL 的功能。本章介绍通过 REST 和 Spring Data 两种方式访问 Elasticsearch。

**第 14 章：** 介绍 Spring Boot Cache，并重点介绍 Redis 作为分布式缓存的实现。在此基础上，改进了 Redis 分布式缓存，通过较少的代码实现了一个具备一二级缓存的技术方案。

**第 15 章：** Spring Boot 应用水平扩展，需实现无会话状态技术，Spring Session 提供了分布式会话管理，本章介绍了 Nginx 作为反向代理的内容，以及 Spring Session 的 Redis 实现及其源码分析。

**第 16 章：** 基于第 15 章 Spring Boot 应用水平扩展技术必然带来分布式协调要求，ZooKeeper 是一个广泛使用的分布式协调器。本章介绍 ZooKeeper 的安装和使用，对领导选取、分布式锁和服务注册三个常用功能做了重点描述，并在 Spring Boot 应用中采用 Curator 来完成这三个功能。

**第 17 章：** Spring Boot 提供了内置监控功能，使得用户通过 HTTP 请求就能知晓服务器的健康状态，如数据源是否可用、NoSQL 服务是否可用、最近的 HTTP 访问的内容等监控信息。本章讲述了线程栈、内存、在线日志、HTTP 访问、RequestMapping 等常用监控功能。其中还讲述通过 dump 线程栈和内存来解决 Spring Boot 应用中的一些性能故障。

# 如何阅读本书

笔者作为一个从事 Java 开发 17 年的程序员，这里给新手一些诚恳的建议，用于帮助新手掌握 Spring Boot 2.0。

建议按照本书每章的例子先模仿一遍，不要急于按照自己的项目要求去改，否则很容易知识掌握不牢固、不全面。如果遇到自己暂时无法理解的知识，也建议优先记住这些知识点。

理解完书中的知识，能运行书中提供的例子（推荐手写，或者从官网下载例子）后，可以尝试主动制造一些错误。看看 Spring Boot 会给你什么样的错误提示，以 1.4.5 节例子为例，如果去掉@RequestMapping 注解，或者如果写成 value="/usercredit/{id123}" 会怎么样，甚至将 getCreditLevel 改成 getCreditLevelTest 会有什么后果（这个改动没有任何影响）。通过主动制造错误，观察 Spring Boot 应用的错误信息来深入学习 Spring Boot。实际上，不仅仅是学习 Spring Boot，这也是学习其他框架，甚至是编程语言或者其他任何编程技术的一种窍门。

本书每章都会提及 Spring Boot 框架的一些接口或者关键类，不了解这些类的实现细节，你

仍然可以运用 Spring Boot，如果想深入掌握 Spring Boot，建议阅读这些类的源代码以了解这些类的职责以及如何实现职责。可以通过 IDE 的快捷键打开这些类，以 Eclipse 为例，用快捷键 Ctrl+T 打开这些类去阅读 Spring 源码，还可以在这些类的方法中打上断点，在运行本书例子的时候，查看在断点处发生了什么来帮助你理解 Spring Boot。比如在第 14 章中使用 Redis 实现分布式缓存的时候，提到了 RedisCacheManager，你可以阅读这个类的源码，并在关键的 getCache 方法上打上断点，观察如何实现 Redis 缓存。

如果对于这些类还是无法理解，则可以通过搜索引擎搜索这些类，总有些博客和技术文章在讨论这些类的职责和实现方式。

谨慎对待互联网搜索结果，这是因为 Spring Boot 2.0 技术本身较为新，发展也较快，通过互联网搜索结果需要关注一下文章的发布日期、文章适用的版本，如果你在使用 Spring Boot 2.0 中遇到任何问题，也欢迎到社区进行交流，社区地址是 ibeetl.com。

最后，请购买正版书籍，鼓励作者和出版社出版更好的技术书籍。

## 致谢

首先感谢我的妻子苗珺对我写书的大力支持，没有她的支持，我是不可能完成一本书的写作的。还有我的儿子，知道我正在做一件很重要的事情的时候就不再让我陪他聊天。

还要感谢电子工业出版社的编辑同志，给予我绝对的信任和支持，并对本书的出版做了非常多的指导，感谢你们付出的辛勤汗水。

最后要感谢的是公司和集团领导对我写书的大力支持，特别是在我需要帮助的时候委派左丽娜同事完成部分章节的编写，没有公司领导的支持，对于中国程序员来说，写一本书几乎是不可能完成的任务。

本书是我写的第一本书，由于 Spring 和 Spring Boot 技术体系博大精深，而我技术有限，写作过程中精力也有限，难免有纰漏，敬请读者指正。

东方邦信金融科技有限公司　李家智（闲大赋）

## 读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32825

# 目录


第 1 章　Java EE 简介 .......... 1

1.1　Java EE .......... 1

1.1.1　Java EE 架构 .......... 1

1.1.2　Java EE 的缺点 .......... 3

1.2　Spring .......... 4

1.2.1　Spring IoC 容器和 AOP .......... 4

1.2.2　Spring 的缺点 .......... 7

1.3　Spring Boot .......... 8

1.4　Hello，Spring Boot .......... 9

1.4.1　创建一个 Maven 工程 .......... 10

1.4.2　增加 Web 支持 .......... 10

1.4.3　Hello Spring Boot 示例 .......... 13

1.4.4　使用热部署 .......... 15

1.4.5　添加 REST 支持 .......... 16

第 2 章　Spring Boot 基础 .......... 17

2.1　检查 Java 环境与安装 Java .......... 17

2.2　安装和配置 Maven .......... 19

2.2.1　Maven 介绍 .......... 20

2.2.2　安装 Maven .......... 22

2.2.3　设置 Maven .......... 23

2.2.4　使用 IDE 设置 Maven .......... 23

2.2.5　Maven 的常用命令 .......... 24

2.3　Spring 核心技术 .......... 27

2.3.1 Spring 的历史 ........ 27
2.3.2 Spring 容器介绍 ........ 28
2.3.3 Spring AOP 介绍 ........ 33

第 3 章 MVC 框架 ........ 37
3.1 集成 MVC 框架 ........ 38
3.1.1 引入依赖 ........ 38
3.1.2 Web 应用目录结构 ........ 38
3.1.3 Java 包名结构 ........ 39
3.2 使用 Controller ........ 40
3.3 URL 映射到方法 ........ 41
3.3.1 @RequestMapping ........ 41
3.3.2 URL 路径匹配 ........ 42
3.3.3 HTTP method 匹配 ........ 43
3.3.4 consumes 和 produces ........ 44
3.3.5 params 和 header 匹配 ........ 45
3.4 方法参数 ........ 46
3.4.1 PathVariable ........ 47
3.4.2 Model&ModelAndView ........ 48
3.4.3 JavaBean 接受 HTTP 参数 ........ 50
3.4.4 @RequsetBody 接受 JSON ........ 52
3.4.5 MultipartFile ........ 53
3.4.6 @ModelAttribute ........ 55
3.4.7 @InitBinder ........ 56
3.5 验证框架 ........ 56
3.5.1 JSR-303 ........ 56
3.5.2 MVC 中使用@Validated ........ 58
3.5.3 自定义校验 ........ 59
3.6 WebMvcConfigurer ........ 60
3.6.1 拦截器 ........ 61
3.6.2 跨域访问 ........ 62
3.6.3 格式化 ........ 63
3.6.4 注册 Controller ........ 64

3.7 视图技术 ........ 64
3.7.1 使用 Freemarker ........ 64
3.7.2 使用 Beetl ........ 66
3.7.3 使用 Jackson ........ 67
3.7.4 Redirect 和 Forward ........ 68
3.8 通用错误处理 ........ 69
3.9 @Service 和@Transactional ........ 72
3.9.1 声明一个 Service 类 ........ 72
3.9.2 事务管理 ........ 73
3.10 curl 命令 ........ 74

**第 4 章 视图技术 ........ 77**

4.1 Beetl 模板引擎 ........ 77
4.1.1 安装 Beetl ........ 78
4.1.2 设置定界符号和占位符 ........ 78
4.1.3 配置 Beetl ........ 79
4.1.4 groupTemplate ........ 79
4.2 使用变量 ........ 80
4.2.1 全局变量 ........ 80
4.2.2 局部变量 ........ 81
4.2.3 共享变量 ........ 81
4.2.4 模板变量 ........ 82
4.3 表达式 ........ 82
4.3.1 计算表达式 ........ 82
4.3.2 逻辑表达式 ........ 83
4.4 控制语句 ........ 83
4.4.1 循环语句 ........ 83
4.4.2 条件语句 ........ 85
4.4.3 try catch ........ 86
4.5 函数调用 ........ 87
4.6 格式化函数 ........ 87
4.7 直接调用 Java ........ 88
4.8 标签函数 ........ 89

4.9 HTML 标签 ........ 90
4.10 安全输出 ........ 91
4.11 高级功能 ........ 91
4.11.1 配置 Beetl ........ 91
4.11.2 自定义函数 ........ 93
4.11.3 自定义格式化函数 ........ 94
4.11.4 自定义标签函数 ........ 95
4.11.5 自定义 HTML 标签 ........ 97
4.11.6 布局 ........ 98
4.11.7 AJAX 局部渲染 ........ 100
4.12 脚本引擎 ........ 101
4.13 JSON 技术 ........ 102
4.13.1 在 Spring Boot 中使用 Jackson ........ 102
4.13.2 自定义 ObjectMapper ........ 103
4.13.3 Jackson 的三种使用方式 ........ 103
4.13.4 Jackson 树遍历 ........ 104
4.13.5 对象绑定 ........ 105
4.13.6 流式操作 ........ 106
4.13.7 Jackson 注解 ........ 107
4.13.8 集合的反序列化 ........ 111
4.14 MVC 分离开发 ........ 113
4.14.1 集成 WebSimulate ........ 113
4.14.2 模拟 JSON 响应 ........ 114
4.14.3 模拟模板渲染 ........ 114

第 5 章 数据库访问 ........ 116
5.1 配置数据源 ........ 116
5.2 Spring JDBC Template ........ 118
5.2.1 查询 ........ 119
5.2.2 修改 ........ 121
5.2.3 JdbcTemplate 增强 ........ 122
5.3 BeetlSQL 介绍 ........ 123
5.3.1 BeetlSQL 功能概览 ........ 124

5.3.2 添加 Maven 依赖 ........ 124
5.3.3 配置 BeetlSQL ........ 125
5.3.4 SQLManager ........ 126
5.3.5 使用 SQL 文件 ........ 127
5.3.6 Mapper ........ 129
5.3.7 使用实体 ........ 131
5.4 SQLManager 内置 CRUD ........ 131
5.4.1 内置的插入 API ........ 131
5.4.2 内置的更新（删除）API ........ 132
5.4.3 内置的查询 API ........ 132
5.4.4 代码生成方法 ........ 133
5.5 使用 sqlId ........ 134
5.5.1 md 文件命名 ........ 134
5.5.2 md 文件构成 ........ 135
5.5.3 调用 **sqlId** ........ 135
5.5.4 翻页查询 ........ 137
5.5.5 TailBean ........ 138
5.5.6 ORM 查询 ........ 139
5.5.7 其他 API ........ 141
5.5.8 Mapper 详解 ........ 142
5.6 BeetlSQL 的其他功能 ........ 143
5.6.1 常用函数和标签 ........ 144
5.6.2 主键设置 ........ 145
5.6.3 BeetlSQL 注解 ........ 147
5.6.4 NameConversion ........ 148
5.6.5 锁 ........ 148

第 6 章 Spring Data JPA ........ 150
6.1 集成 Spring Data JPA ........ 150
6.1.1 集成数据源 ........ 150
6.1.2 配置 JPA 支持 ........ 151
6.1.3 创建 Entity ........ 152
6.1.4 简化 Entity ........ 154

6.2 Repository ........ 155
6.2.1 CrudRepository ........ 155
6.2.2 PagingAndSortingRepository ........ 156
6.2.3 JpaRepository ........ 156
6.2.4 持久化 Entity ........ 157
6.2.5 Sort ........ 158
6.2.6 Pageable 和 Page ........ 159
6.2.7 基于方法名字查询 ........ 160
6.2.8 @Query 查询 ........ 162
6.2.9 使用 JPA Query ........ 163
6.2.10 Example 查询 ........ 166

第 7 章 Spring Boot 配置 ........ 167
7.1 配置 Spring Boot ........ 167
7.1.1 服务器配置 ........ 167
7.1.2 使用其他 Web 服务器 ........ 168
7.1.3 配置启动信息 ........ 170
7.1.4 配置浏览器显示 ico ........ 172
7.2 日志配置 ........ 172
7.3 读取应用配置 ........ 174
7.3.1 Environment ........ 175
7.3.2 @Value ........ 175
7.3.3 @ConfigurationProperties ........ 176
7.4 Spring Boot 自动装配 ........ 177
7.4.1 @Configuration 和@Bean ........ 177
7.4.2 Bean 条件装配 ........ 178
7.4.3 Class 条件装配 ........ 179
7.4.4 Environment 装配 ........ 179
7.4.5 其他条件装配 ........ 180
7.4.6 联合多个条件 ........ 180
7.4.7 Condition 接口 ........ 181
7.4.8 制作 Starter ........ 183

第 8 章 部署 Spring Boot 应用 ........ 184
8.1 以 jar 文件运行 ........ 184
8.2 以 war 方式部署 ........ 186
8.3 多环境部署 ........ 188
8.4 @Profile 注解 ........ 190

第 9 章 Testing 单元测试 ........ 192
9.1 JUnit 介绍 ........ 192
9.1.1 JUnit 的相关概念 ........ 192
9.1.2 JUnit 测试 ........ 193
9.1.3 Assert ........ 195
9.1.4 Suite ........ 195
9.2 Spring Boot 单元测试 ........ 196
9.2.1 测试范围依赖 ........ 196
9.2.2 Spring Boot 测试脚手架 ........ 196
9.2.3 测试 Service ........ 197
9.2.4 测试 MVC ........ 200
9.2.5 完成 MVC 请求模拟 ........ 201
9.2.6 比较 MVC 的返回结果 ........ 202
9.2.7 JSON 比较 ........ 203
9.3 Mockito ........ 204
9.3.1 模拟对象 ........ 205
9.3.2 模拟方法参数 ........ 206
9.3.3 模拟方法返回值 ........ 208
9.4 面向数据库应用的单元测试 ........ 209
9.4.1 @Sql ........ 209
9.4.2 XLSUnit ........ 211
9.4.3 XLSUnit 的基本用法 ........ 212

第 10 章 REST ........ 218
10.1 REST 简介 ........ 219
10.1.1 REST 风格的架构 ........ 220
10.1.2 使用“api”作为上下文 ........ 220

10.1.3 增加一个版本标识 221
10.1.4 标识资源 221
10.1.5 确定 HTTP Method 221
10.1.6 确定 HTTP Status 223
10.1.7 REST VS. WebService 223
10.2 Spring Boot 集成 REST 224
10.2.1 集成 REST 224
10.2.2 @RestController 224
10.2.3 REST Client 226
10.3 Swagger UI 230
10.3.1 集成 Swagger 230
10.3.2 Swagger 规范 232
10.3.3 接口描述 233
10.3.4 查询参数描述 234
10.3.5 URI 中的参数 235
10.3.6 HTTP 头参数 235
10.3.7 表单参数 235
10.3.8 文件上传参数 236
10.3.9 整个请求体作为参数 236
10.4 模拟 REST 服务 238

第 11 章 MongoDB 240
11.1 安装 MongoDB 240
11.2 使用 shell 241
11.2.1 指定数据库 242
11.2.2 插入文档 243
11.2.3 查询文档 244
11.2.4 更新操作 245
11.2.5 删除操作 246
11.3 Spring Boot 集成 MongoDB 246
11.4 增删改查 247
11.4.1 增加 API 247
11.4.2 根据主键查询 API 248

11.4.3 查询 API ........ 249
11.4.4 修改 API ........ 250
11.4.5 删除 API ........ 251
11.4.6 使用 MongoDatabase ........ 251
11.4.7 打印日志 ........ 253

第 12 章 Redis ........ 254
12.1 安装 Redis ........ 254
12.2 使用 redis-cli ........ 255
12.2.1 安全设置 ........ 256
12.2.2 基本操作 ........ 256
12.2.3 keys ........ 257
12.2.4 Redis List ........ 258
12.2.5 Redis Hash ........ 260
12.2.6 Set ........ 261
12.2.7 Pub/Sub ........ 262
12.3 Spring Boot 集成 Redis ........ 264
12.4 使用 StringRedisTemplate ........ 265
12.4.1 opsFor ........ 266
12.4.2 绑定 Key 的操作 ........ 267
12.4.3 RedisConnection ........ 268
12.4.4 Pub/Sub ........ 269
12.5 序列化策略 ........ 270
12.5.1 默认序列化策略 ........ 272
12.5.2 自定义序列化策略 ........ 273

第 13 章 Elasticsearch ........ 276
13.1 Elasticsearch 介绍 ........ 276
13.1.1 安装 Elasticsearch ........ 276
13.1.2 Elasticsearch 的基本概念 ........ 278
13.2 使用 REST 访问 Elasticsearch ........ 279
13.2.1 添加文档 ........ 279
13.2.2 根据主键查询 ........ 281

13.2.3 根据主键更新 ...... 281
13.2.4 根据主键删除 ...... 283
13.2.5 搜索文档 ...... 284
13.2.6 联合多个索引搜索 ...... 287
13.3 使用 RestTemplate 访问 ES ...... 288
13.3.1 创建 Book ...... 288
13.3.2 使用 RestTemplate 获取搜索结果 ...... 288
13.4 Spring Data Elastic ...... 290
13.4.1 安装 Spring Data ...... 290
13.4.2 编写 Entity ...... 291
13.4.3 编写 Dao ...... 291
13.4.4 编写 Controller ...... 293

第 14 章 Cache ...... 295
14.1 关于 Cache ...... 295
14.1.1 Cache 的组件和概念 ...... 296
14.1.2 Cache 的单体应用 ...... 296
14.1.3 使用专有的 Cache 服务器 ...... 297
14.1.4 使用一二级缓存服务器 ...... 298
14.2 Spring Boot Cache ...... 299
14.3 注释驱动缓存 ...... 300
14.3.1 @Cacheable ...... 300
14.3.2 Key 生成器 ...... 301
14.3.3 @CachePut ...... 303
14.3.4 @CacheEvict ...... 304
14.3.5 @Caching ...... 305
14.3.6 @CacheConfig ...... 305
14.4 使用 Redis Cache ...... 305
14.4.1 集成 Redis 缓存 ...... 305
14.4.2 禁止缓存 ...... 306
14.4.3 定制缓存 ...... 306
14.5 Redis 缓存原理 ...... 307
14.6 实现 Redis 两级缓存 ...... 309

14.6.1 实现 TwoLevelCacheManager ........ 309
14.6.2 创建 RedisAndLocalCache ........ 310
14.6.3 缓存同步说明 ........ 313
14.6.4 将代码组合在一起 ........ 314

第 15 章 Spring Session ........ 316

15.1 水平扩展实现 ........ 316
15.2 Nginx 的安装和配置 ........ 318
15.2.1 安装 Nginx ........ 318
15.2.2 配置 Nginx ........ 319
15.3 Spring Session ........ 321
15.3.1 Spring Session 介绍 ........ 321
15.3.2 使用 Redis ........ 322
15.3.3 Nginx+Redis ........ 324

第 16 章 Spring Boot 和 ZooKeeper ........ 326

16.1 ZooKeeper ........ 327
16.1.1 ZooKeeper 的数据结构 ........ 327
16.1.2 安装 ZooKeeper ........ 328
16.1.3 ZooKeeper 的基本命令 ........ 329
16.1.4 领导选取演示 ........ 332
16.1.5 分布式锁演示 ........ 333
16.1.6 服务注册演示 ........ 333
16.2 Spring Boot 集成 ZooKeeper ........ 334
16.2.1 集成 Curator ........ 335
16.2.2 Curator API ........ 336
16.3 实现分布式锁 ........ 338
16.4 服务注册 ........ 341
16.4.1 通过 ServiceDiscovery 注册服务 ........ 341
16.4.2 获取服务 ........ 342
16.5 领导选取 ........ 343

第 17 章 监控 Spring Boot 应用 ........ 345
17.1 安装 Acutator ........ 346
17.2 HTTP 跟踪 ........ 347
17.3 日志查看 ........ 348
17.4 线程栈信息 ........ 350
17.5 内存信息 ........ 352
17.6 查看 URL 映射 ........ 355
17.7 查看 Spring 容器管理的 Bean ........ 355
17.8 其他监控 ........ 356
17.9 编写自己的监控信息 ........ 357
17.9.1 编写 HealthIndicator ........ 357
17.9.2 自定义监控 ........ 358

# 第1章 Java EE简介

## 1.1 Java EE

我从 2001 年开始接触 Java EE 技术，当时面对 Java EE 那么多组件和规范，比如，EJB 技术，确实有点发懵了。编写一个企业应用居然用到了那么多技术，曾经的电信项目，启动就需要 10 分钟，每次发布都需要一个小时。作为新手的我是不能理解的，这也是当时大多数程序员的心态。然而 Java EE 针对复杂企业系统所指定的规范和实现，能满足复杂的企业应用需求，这也是为什么 Java EE 很快就流行起来，并在电信、银行等领域广泛使用的原因。2003 年 Spring 横空出世，告诉所有人，编写企业应用、Web 应用，并非需要全部的 Java EE 技术，也不需要像 EJB 那样复杂的使用方式和部署方式。使用 Spring，开发并部署网站和企业应用变得很便捷。同时 Spring 建立在 Java EE 基础上，同样可以使用 Java EE 的功能。基于技术的创新和兼容传统 Java EE 技术，使得 Spring 框架很快流行起来，普遍使用在传统企业应用和互联网应用中。

### 1.1.1 Java EE 架构

要应用 Spring Boot 技术，并不一定需要先从 Spring 技术开始，更不需要了解 Java EE。然而，稍微了解 Java EE 和 Spring 技术，对 Spring Boot 会有更深的理解。

从根本上来说，Java EE 是一种企业应用的软件架构。在了解它之前，让我们来先看看它的发展过程，它的发展过程总是与分布式应用和互联网应用密切相关。

Java EE 与 Web：互联网从根本上改变了对企业软件的系统需求，软件需要处理来自互联网的大量请求，并要及时做出响应。

Java EE 与分布式应用：20 世纪 80 年代，个人计算机性价比逐渐达到了高端工作站和服务器的水准，使分布式计算应用迅速普及。SUN 在推出 Java 后，紧接着推出了远程方法调用 RMI，并在 90 年代末期，以 RMI 为通信基础构建了 Java EE。在相当长的一段时间里，Java EE 就是一种分布式应用，这让 Java EE 战胜了 CORBA、.COM+，但是也带来了巨大的系统交互开销（超出一个数量级的）。毕竟不是所有的企业应用和互联网应用都是分布式的，这让一些人认为 Java EE 架构有问题。然而这并不是 Java EE 的错，在市场上，Java EE 需要迎合当时的分布式技术潮流。现在如果你不用分布式，Java EE 也同样提供了相关技术供你使用，如 Local EJB。

作为企业应用或者互联网应用的架构，总有如下功能需求，Java EE 有相应的规范实现与之对应。

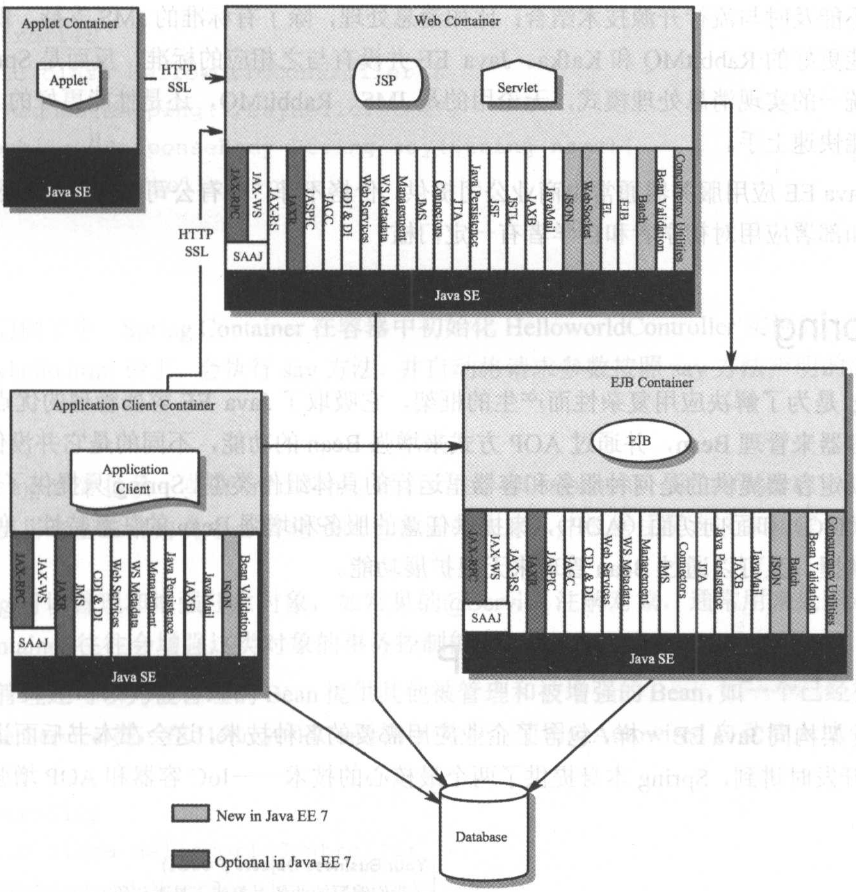
- Web 支持：企业应用、互联网应用越来越多的是基于 B/S 的结构，Java EE 对应的有 Servlet 规范，规定了 Web 容器、Servlet 组件，还设有 JSP&JSTL 处理动态页面。
- 事务支持：提供事务管理器，支持管理事务，如单一数据库、多个数据库，以及数据库和其他资源的事务协作等。Java EE 里提供了 JTA 事务 API 和 JTS 事务服务规范。事务支持也实现了分布式事务管理，管理多个数据库或者支持事务的资源。
- 消息服务：企业各个系统、系统模块之间通过消息服务进行并步交互，Java EE 提供了 JMS 服务，用于系统间可靠的消息交互。
- 数据库持久层、Java EE 先有 EJB 规范，后来又提出了更有实际操作性的 JPA，这些都是企业访问数据库常用的方法。
- Container：提供了 WebContainer，用于实现 Servlet，以及 EJB Container，实现 EJB，Container 用于管理这些组件，并提供组件需要的服务，比如 JTS、JMS 等。

其他技术还包括如下所述的内容。

- JWS：这也是分布式系统交互的一种方式，是 Java 实现的一种 WebService。
- JAX-RS：Java EE 6 引入的新技术，通过 REST 进行交互。
- JNDI：查找服务和对象的接口，如查找一个配置好的数据源。
- JAXP/JAXB：XML 流行的时候，解析和绑定 Java Bean 的 XML 工具。
- JAX-RPC：分布式系统交互的一种方式，通过 RPC 方式调用。
- JACC：安全认证。

- Java EE Application Server：提供了上述所有服务的 Web 服务器，我们常用的 Tomcat、Resin，只能算是 Java EE Web 服务器，开源的 JBoss、商业的 weblogic 等称为应用服务器。

综上所述，Java EE 提供了企业应用技术的实现规范，简化了企业应用和 Web 开发，如下图所示。




## 1.1.2 Java EE 的缺点

Java EE 的出现，结束了 Web 开发的技术无序状态，让程序员、架构师用同一种思维去思考如何架构和开发应用（指企业应用和 Web 网站，以下同），然而 Java EE 也有其局限性。

- 过于复杂：Java EE 针对的是复杂的分布式企业应用，然而现实情况是大部分应用都比较简单，复杂的架构带来了复杂的开发方式和部署方式（早期的 Java EE 普通系统打

包部署都需要数十分钟）。

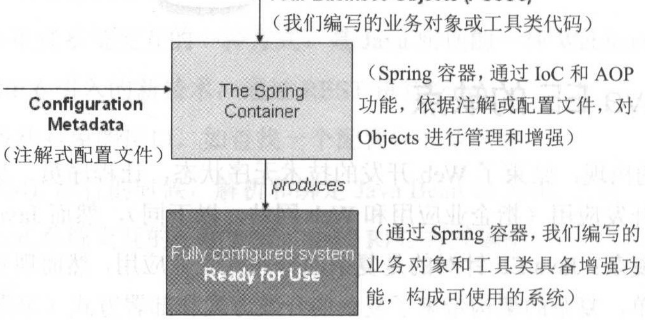
- 追求分布式：大部分应用并非都是 Java EE 假定的分布式系统，像 EJB、JMS、JWS 等技术实现门槛又高又容易出错。Spring 曾经反对过这种分布式架构，并只提供了容器管理，因此获得了巨大成功。大型应用采用分布式架构不可避免，Spring 提供了其他的技术支持，比如支持 RESTFul 架构。
- 不能及时与流行开源技术结合：比如消息处理，除了有标准的 JMS 支持，现在还有性能更好的 RabbitMQ 和 Kafka。Java EE 并没有与之相应的标准，反而是 Spring，具有统一的实现消息处理模式，无论用的是 JMS、RabbitMQ，还是性能更好的 Kafka，都能快速上手。
- Java EE 应用服务器通常由商业公司提供，价格不菲，少有公司采用。管理应用服务器和部署应用对初学者和自学者有一定门槛。

## 1.2 Spring

Spring 是为了解决应用复杂性而产生的框架，它吸取了 Java EE 容器管理的优点，提供了一个 IoC 容器来管理 Bean，并通过 AOP 方式来增强 Bean 的功能，不同的是它并没像 Java EE 那样详细规定容器提供的是何种服务和容器里运行的具体组件类型。Spring 只提供了两种机制，控制反转（IoC）和面向切面（AOP），来提供任意的服务和增强 Bean 的任意特性，使得 Spring 本身容易掌握，又可以通过 Bean 管理来无限扩展功能。

### 1.2.1 Spring IoC 容器和 AOP

Spring 架构同 Java EE 一样，包含了企业应用需要的各种技术，这会在本书后面讲解 Spring Boot 应用开发时讲到，Spring 本身提供了两个最核心的技术——IoC 容器和 AOP 增强，如下图所示。



关于 IoC 和 AOP 的详细说明如下所述。

**IoC Core Container**：Spring Container 负责管理你的任意对象，并结合你对对象的描述进行初始化和加强。比如，对于一个用注解@Controller 声明的对象，Spring 会认为这个对象是个 Web Controller，如果这个对象里的方法有@RequestMapping 注解，则会将客户端发起的 HTTP 请求转化成 Java 方法调用。

```
@Controller
public class HelloworldController {
    @RequestMapping("/sayhello.html")
    public @ResponseBody String say(String name){
        return "hello "+name;
    }
}
```

上面的例子中，Spring Container 在容器中初始化 HelloworldController 实例后，对于客户端发起的/sayhello.html 请求，会执行 say 方法，并自动将请求参数按照 say 方法声明的名称一一对应上。

> Spring 通常提供一些@Controller、@Service、@Component、@Configuration 注解，只有使用这些注解的类才会引起Spring容器的注意，并根据注解含义来管理和增强对象。

Spring 可以管理和增强任意对象，如常见的@Service 注解对象，通常用来处理业务逻辑，Spring Container 往往会增强这类对象的事务控制能力。

容器管理还可以为被管理的 Bean 提供其他被管理和被增强的 Bean，如一个已经被@Service 注解的 UserService 类，在 HelloworldController 类中，使用@Autowired 自动注入这个实例：

```
@Controller
public class HelloworldController {
    @Autowired UserService userService;
}
```

**AOP**：上面提到的对象增强离不开 AOP 技术，AOP（Aspect Oriented Programming）指面向切面编程，通过预编译方式或者运行时刻对目标对象动态地添加功能。AOP 分离了企业应用的业务逻辑和系统级服务，比如事务服务，还有应用系统的审计、安全访问等代码。比如要实现用户访问控制，可以对每个 Controller 的方法使用一个自定义的注解 Function，用 Spring AOP 向 Controller 每个方法动态地添加用户权限校验功能，类似如下：

```
@RequestMapping("/sayhello.html")
public @ResponseBody String say(String name){
  return "hello "+name;
}

@RequestMapping("/adduser.html")
@Function("user.add")
public @ResponseBody String addUser(String name){
  ......
}
```

注解 Function 是自定义一个注解，接受一个字符串，表示 Controller 方法对应的业务功能。用户是否能访问“user.add”功能，将在数据库中配置。

使用 AOP 对所有的 Controller 进行增强：

```
@Configuration
@Aspect
public class RoleAccessConfig {

    @Around("within(@org.springframework.stereotype.Controller *) &&
@annotation(function)")
    public  Object  functionAccessCheck(final  ProceedingJoinPoint  pjp,
Function function)  throws Throwable {
        if(function!=null){
            String functionName = function.value();
            if(!canAccess(functionName)){
                MethodSignature ms = (MethodSignature) pjp.getSignature();
                throw new RuntimeException("Can not Access "+ms.getMethod());
            }
        }
        // 继续处理原有调用
        Object o = pjp.proceed();
        return o;
    }

    protected boolean canAccess(String functionName){
```

```
        if(functionName.length()==0){
            // 总是允许访问
            return true;
        }else{
    // 取出当前用户对应的所有角色，从数据库中查询角色是否有访问 functionName 的权限
            return false ;
        }
    }
}
```

这段代码比较复杂，可以按照这个顺序理解：

- @Configuration 注解成功引起 Spring 容器的注意；
- @Aspect 让 Spring 容器知道，这是一个 AOP 类；
- @Around 是 AOP 的一种具体方式，我们将在下一章详细介绍，这里只需要知道，它能对目标方法调用前和调用后进行处理；
- within(@org.springframework.stereotype.Controller*)可以理解为对所有使用@Controller 注解的类进行 AOP；
- @annotation(function) 表示另外一个条件，也就是对具有 function 参数对应的注解方法进行 AOP；
- functionAccessCheck 是实现 AOP 的具体代码。

关于 AOP，不会在这里做过多说明，如果不理解，也没关系，我们将在下一章详细说明。这里举例只是说明 Spring 通过 AOP 来增强 Bean 的功能特性。

## 1.2.2 Spring 的缺点

随着 Spring 的功能越来越强，在使用 Spring 的时候，门槛也变得高了起来，诸如搭建一个基于 Spring 的 Web 程序却并不简单，需要进行各种配置。Spring 中一些过时的技术也经常无意地被引用到最新项目中。新手面对同一种需求会面临选择困难，比如处理事务是用 XML 配置好，还是使用注解@Trasactional 好？这是因为 Spring 通过多年的发展，本身变得臃肿，过时的使用方式没有被淘汰固然很好，但会给初学者带来使用上的混乱。同时 Spring 也集成了越来越多的第三方技术，如何方便地使用这些第三方技术，各版本之间不会产生冲突也需要一定的实践。尽管 Spring 很强大，但它也在犯 Java EE 的错误，例如有如下缺点：

- 使用门槛升高，要入门 Spring 需要较长时间。
- 对过时技术兼容，导致使用复杂度升高。
- XML 配置已经不是流行的系统配置方式。
- 集成第三方工具时候，程序员还要考虑工具之间的兼容性。
- 系统启动慢，不具备热部署功能，完全依赖虚拟机或者 Web 服务器的热部署。

庆幸的是，Spring 的开发者们及时认识到了这个问题，推出了基于 Spring 技术的 Spring Boot，解决了 Spring 的如上问题，尤其是上手难、技术使用不统一这两个缺点。经过快速发展，Spring Boot 也逐渐被开发人员所喜爱，成为搭建系统非常好的开发框架。正如本书一直强调的，Spring Boot 能极速开发 Web 系统，也能容易地架构大的分布式系统。

## 1.3 Spring Boot

Spring Boot 简化了 Spring 应用开发，不需要配置就能运行 Spring 应用，Spring Boot 管理 Spring 容器、第三方插件，并提供很多默认系统级的服务。大部分 Spring 应用，无论是简单的 Web 系统，还是构建复杂系统，都只需要少量配置和代码就能完成。这有点像每个公司基于 Spring 框架做的内部开发框架，不同的是，Spring Boot 更完善、更强大。

Spring Boot 通过 Starter 来提供系统级服务，Spring Boot 已经提供了一系列 Starter，你也可以开发自己的 Starter。比如需要开发一个 Web 应用，只要在 pom.xml 中（如果还不了解 Maven 的 pom，可以参考第 2 章关于 Maven 的介绍）声明一下即可。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

如果你的应用中用到了 Redis，则使用 spring-boot-starter-data-redis，Spring Boot 会自动为你配置好 Spring 需要的各种配置文件、Redis 的 jar 包、依赖包，以及合适的版本，下表是 Spring Boot 提供的常用 Starter。

| 名 称 | 作 用 |
| --- | --- |
| spring-boot-starter-web | Web 开发支持，默认使用 Tomcat8 |
| spring-boot-starter-aop | AOP 开发支持，使用 AspectJ |
| spring-boot-starter-jdbc | Spring JDBC |

续表

| 名　　称 | 作　　用 |
|---|---|
| spring-boot-starter-data-jpa | JPA 方式访问数据库，使用 Hibernate 作为 JPA 实现 |
| spring-boot-starter-data-elasticsearch | 集成 Elasticsearch，默认访问 localhost:9200 |
| spring-boot-starter-data-redis | 集成 Redis，使用 JRedis，默认连接 localhost:6379 |
| spring-boot-starter-cache | 缓存，支持多种缓存方式，如本地的、Redis、Ehcache 等 |
| spring-boot-devtools | 应用程序快速重启的工具，提升开发体验 |
| spring-boot-starter-data-mongodb | 集成 MongoDB，默认访问 mongodb://localhost/test |
| spring-boot-starter-data-neo4j | 集成 neo4j，默认访问 localhost:7474 |
| spring-boot-starter-data-gemfire | 集成分布式缓存 |
| spring-boot-starter-data-solr | 基于 Apache lucene 的搜索平台，默认访问 http://localhost:8983/solr |
| spring-boot-starter-data-cassandra | 集成 Cassandra，默认访问 localhost:7474 |
| spring-boot-starter-data-ldap | 集成 ldap |
| spring-boot-starter-activemq | 消息集成 ActiveMQ 支持 |
| spring-boot-starter-amqp | 消息集成 AMQP 协议支持，如支持 RabbitMQ |
| spring-boot-starter-jta-atomikos | 分布式事务支持，使用 atomikos |
| spring-boot-starter-jta-bitronix | 一个开源的分布式事务支持 |
| spring-boot-starter-test | 包含 JUnit、Spring Test、Hamcrest、Mockito 等测试工具 |
| spring-boot-starter-webservices | webservice 支持 |
| spring-boot-starter-websocket | websocket 支持 |
| spring-boot-starter-jersey | REST 应用和 Jersey 支持 |
| spring-boot-starter-freemarker | Freemaker 支持 |

相比于 Spring，Spring Boot 具有以下优点：

- 实现约定大于配置，是一个低配置的应用系统框架。不像 Spring 那样“地狱般的配置体验”，Spring Boot 不需要配置或者极少配置，就能使用 Spring 大量的功能。
- 提供了内置的 Tomcat 或者 Jetty 容器。
- 通过依赖的 jar 包管理、自动装配技术，容易支持与其他技术体系、工具集成。
- 支持热加载，开发体验好。也支持 Spring Boot 系统监控，方便了解系统运行状况。

## 1.4 Hello，Spring Boot

本节完成一个简单的 Web 应用，输出“Hello，Spring Boot”。例子基于已经安装好的环境，只需要 JDK8、Eclipse，还有 Maven3 就可以了，如果你还没有安装或者熟悉这些工具，仍然可以先阅读本章，也可以先参考第 2 章的环境搭建后，再按照本章的代码边写边学习。手敲代码

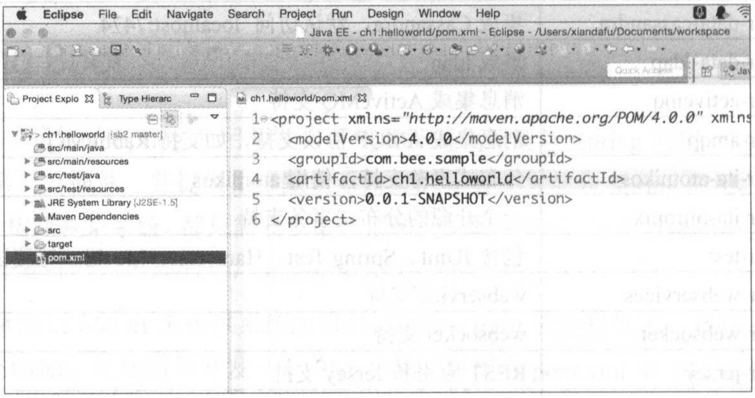
是最快的一种学习方式。

### 1.4.1 创建一个Maven工程

打开 Eclipse，选择 File→New→Maven Project，在弹出的面板里，勾选“create a simple project(skip archetype selection)”，并选择下一步。我们将从一个空白的 maven 工程开始来创建 spring 应用。

在下一个面板里，需要输入 group Id 和 Artifact Id，前者输入 com.bee.sample，后者输入 ch1.helloworld，然后点击“完成”按钮。

这时候我们就能看到一个 Maven 工程出现在 Eclipse 里，如下图所示。




打开 pom 文件，会看到 Eclipse 已经为我们创建了一个 Maven 工程的基本信息，接下来，我们需要在 pom 文件中增加 Spring Boot 的依赖。

### 1.4.2 增加Web支持

本章的例子是一个简单的 Web 应用，不用像以往的各种框架需要下载安装一个 Web 服务器，然后创建 Web 工程，配置 web.xml、applicationContext.xml。使用 Spring Boot，仅仅需要在 pom 文件中声明使用 Spring Boot，并添加一个 spring-boot-starter-web 的依赖即可。Spring Boot 会使用内置的 Tomcat 作为 Web Server，并且自动配置好 Spring 应用所需要的一切配置文件。

打开 pom.xm，添加如下内容，使工程变成 Spring Boot 应用：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
```

```
  <version>2.0.0.M4</version>
</parent>
```

因为我们搭建的是 Web 应用，必须添加 spring-boot-starter-web 依赖，因此增加如下内容：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

spring-boot-dependencies 默认会使用内置的 Tomcat，并支持 Spring MVC、RESTFul 服务。

做好上面的配置后，整个 Maven 文件看起来是这个样子的：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.bee.sample</groupId>
  <artifactId>ch1.helloworld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
  <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.M3</version>
  </parent>
  <!-- Add typical dependencies for a web application -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
  <!-- 参考下面的说明 -->
  <repositories></repositories>
  <pluginRepositories></pluginRepositories>
```

```
</project>
```

> **注意：**使用 Maven 的唯一问题在于有可能 Maven 自动下载依赖包太慢，我们需要做的是耐心等待，或者修改 Maven 的镜像，改为国内镜像。关于使用 Maven 正确使用方法，请参考本书第 2 章。

本书在写作的时候，Spring Boot 2.0 还并没有正式发布，处于里程碑阶段 3（Milestone 3），因此，你可以将本书中的 Spring Boot 版本改成 Spring Boot 2.0 的正式版本号，比如改成：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
</parent>
```

如果你在阅读本书的时候，Spring Boot 2.0 还处于里程碑阶段，或者 RC 阶段（release candidate，正式版候选），那么 Spring Boot 2.0 并未在中央仓库中，因此 pom 还需要指明 Spring Boot 2.0 的仓库位置，需要添加如下内容：

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/snapshot</url>
```

```
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
  </pluginRepository>
</pluginRepositories>
```

> 本书所有的例子都可以通过扫描封面勒口的二维码获得下载地址，或者直接访问 ibeetl.com 获得源码下载地址。

### 1.4.3 Hello Spring Boot 示例

准备好 pom 后，接下来在工程中创建一个有 main 方法的类，包名是 com.bee.sample.ch1，类名是 Ch1Application，如下所示。

```
package com.bee.sample.ch1;
public class Ch1Application {
  public static void main(String[] args) {

  }
}
```

这只是一个 Java 普通的类，我们现在需要使其成为一个 Spring Boot 应用，首先在 Ch1Application 类上添加注解：

```
@SpringBootApplication
public class Ch1Application {
}
```

然后在 main 方法中添加一行：

```
SpringApplication.run(Ch1Application.class, args);
```

整个代码看起来是这个样子的：

```
package com.bee.sample.ch1;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Ch1Application {

  public static void main(String[] args) {
    SpringApplication.run(Ch1Application.class, args);
  }
}
```

这个类就完全是一个 Spring Boot 应用，可以运行。因为还没有写 Controller，还不能通过浏览器访问应用，因此再创建一个类，类名为 HelloworldController，包名是 com.bee.sample.ch1.controller，如下所示。

```
package com.bee.sample.ch1.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloworldController {
  @RequestMapping("/say.html")
  public @ResponseBody String say(){
    return "Hello Spring Boot";
  }
}
```

> **注意：** 对于 Spring Boot 应用，建议启动程序的包名层次最高，其他类均在其下，这样 Spring Boot 默认自动搜索启动程序之下的所有类。

- @Controller 是 Spring MVC 注解，表示此类用于负责处理 Web 请求。
- @RequestMapping 是 Spring MVC 注解，表示如果请求路径匹配，被注解的方法将被调用。
- @ResponseBody 表示此方法返回的是文本而不是视图名称。

编写完这个类，重新启动 Ch1Application 程序，这个时候，我们会看到启动日志里有如下

内容：

```
Mapped "{[/say.html]}" onto public java.lang.String
com.bee.sample.ch1.controller.HelloworldController.say()
.....
Tomcat started on port(s): 8080 (http)
Started Ch1Application in 2.869 seconds (JVM running for 3.33)
```

打开浏览器，访问地址 http://127.0.0.1:8080/say.html，可以看到输出了“Hello Spring Boot”。

### 1.4.4 使用热部署

在上面的过程中，修改类时必须再次重新运行应用，对于开发者来说非常不方便。Spring Boot 提供了 spring-boot-devtools，它能在修改类或者配置文件的时候自动重新加载 Spring Boot 应用，需要打开 pom 文件，添加如下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
```

然后重启应用，这时修改 say 方法，改成如下：

```
@RequestMapping("/sayhello.html")
```

这时观察 Eclipse 控制台日志输出，发现 Spring Boot 已经检测到 class 文件变化，重启了，输出如下：

```
Mapped "{[/sayhello.html]}" onto public java.lang.String
com.bee.sample.ch1.controller.HelloworldController.say()
....
LiveReload server is running on port 35729
....
Tomcat started on port(s): 8080 (http)
Started Ch1Application in 0.574 seconds (JVM running for 97.496)
```

主要多了两个变化，LiveReload server 用于监控 Spring Boot 应用文件变化，这是因为我们增加了“spring-boot-devtools”，另外启动时间变为 0.5 秒。为什么这么快启动？因为 Spring Boot 再次重启，避免了重启 Tomcat Server，也避免了重启已经加载的 Spring 相关类，只重新加载变化的类。所以速度很快，基本上改完代码或者配置，就能立即进行调试。

## 1.4.5 添加 REST 支持

如果你的系统并非一个单一的 Web 应用，而是由多个系统构成的，比如独立出一个积分系统，为其他系统（如支付系统）提供管理用户积分、显示用户积分、增加用户积分等操作。系统之间的调用方式有很多，RESTFul 也是一种很好的方式。Spring Boot 能很方便地支持 RESTFul 应用。

新创建一个类 UserReditRestController，包名是 com.bee.sampel.ch1.rest：

```
package com.bee.sample.ch1.rest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserReditRestController {
  @RequestMapping(value="/usercredit/{id}")
  public  Integer getCreditLevel(@PathVariable String id) {
    // 模拟 id 用户的信用等级
    return 3;
  }
}
```

此时访问路径为 http://127.0.0.1:8080/usercredit/123，数字 123 是任意的，对应 getCreditLevel 的参数 id。

本例使用了@RestController 而不是@Controller，方法也没有使用@ResponseBody 注解。可以理解为@RestController 相当于@Controller 和@ResponseBody。我们在后面 REST 一章将深入学习 REST 的相关知识。RESTFul 只是一种架构风格，并不是一种特别的技术体系。

> **注意：**对于多个系统互相访问，最好不要直接访问对方的数据库，而应该采用类似 RESTFul 架构，封装了逻辑的接口。这样，对方系统的数据库变更，业务逻辑变化或者版本升级，都不会影响其他系统，第 10 章会介绍 RESTFul 架构和 REST 支持。

# 第 2 章 Spring Boot 基础

本章首先介绍如何安装 Spring Boot 应用的开发环境，如果你是新手，需要安装 Java8 和 Maven3。Spring Boot 应用中 Maven 是必备工具，因此这一章也会详细介绍 Maven 工具的安装、配置和使用。最后会再次介绍 Spring 的历史、IoC 容器和 Spring 常用的注解。如果你已经熟悉 Java 和 Maven，可以直接跳过这一章。

## 2.1 检查 Java 环境与安装 Java

Spring Boot 2.0 需要安装 JDK8 以上版本，可以先进入命令行，通过如下命令查看版本号：

```
java -version
```

如果安装了 JDK8，应该有以下显示：

```
java version "1.8.0_73"
Java(TM) SE Runtime Environment (build 1.8.0_73-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.73-b02, mixed mode)
```

如果显示版本过低，则需要安装 1.8 版本，如果报错：

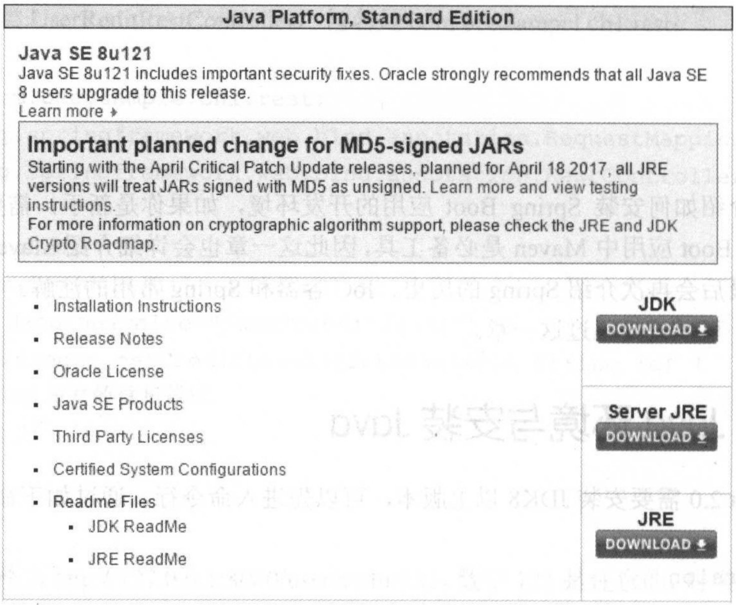
```
'java'不是内部或外部命令，也不是可运行的程序或批处理文件。
```

则说明没有安装 JDK，也需要安装 JDK1.8。

> **注意：**JDK 是指 Java 的开发环境，包含了 Java 运行(JRE)和代码需要的编译、调试、程序诊断两部分。

无论是版本过低还是未安装 JDK8，都可以从官网下载 JDK 进行安装。

- 进入官网 http://www.oracle.com/。
- 或者 Java 下载 http://www.oracle.com/technetwork/indexes/downloads/index.html#java。
- 选择下载 Java SE（如果国外网站下载慢，也可以从国内站点下载），如下图所示。




- 安装 JDK，安装目录最好在专门的 Java 目录下，如笔者的 D:/Java。

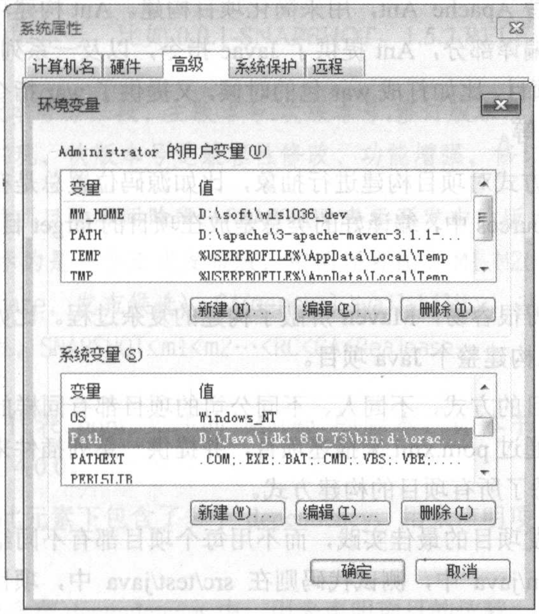
安装完毕后，再次进入命令行：

```
java -version
```

如果显示“'java' 不是内部或外部命令，也不是可运行的程序或批处理文件”，则需要手工设置一下 Path。

点击“我的电脑”，右键选择“属性”，选择“高级系统属性设置”，点击“环境变量”按钮。

找到 Path 变量（如果没有，则新建），添加 Java 的 bin 目录的路径到变量最前面，如笔者的 "D:\Java\jdk1.8.0_73\bin"，如下图所示。




另外找到 JAVA_HOME（如果没有，则新建一个），更改其值为最新的 Java 安装目录，如笔者的 D:\Java\jdk1.8.0_73/。

> **注意：** 现在越来越多的 Java 工具软件不需要依赖 JAVA_HOME 的设置来找到 Java 的运行环境，但设置 JAVA_HOME 还是一个不错的选择。

最后，环境变量值大概类似如下：

```
JAVA_HOME:D:\Java\jdk1.8.0_73
Path:D:\Java\jdk1.8.0_73\bin;d:\oracle\....
```

## 2.2 安装和配置 Maven

Maven 是项目构建工具，能把项目抽象成 POM（project object model），Maven 使用 POM 对项目进行构建、打包、文档化等操作。最重要的是解决了项目需要类库的依赖管理，简化了项目开发环境搭建的过程，使得我们开发一个从简单到大型的复杂项目变得很容易。

### 2.2.1 Maven 介绍

Maven 最初用于代替 Apache Ant，用来简化项目构建。Ant 构建项目采用的是指令方式，比如，构建项目的源码编译部分，Ant 提供了 Javac 指令，以及一系列属性，包括源代码路径、依赖库的路径等。打包项目，比如打成 war 包的时候，又提供了 war 指令和附带的一系列属性，如编译好的 class 的目录等。

Maven 采用了不同方式对项目构建进行抽象，比如源码位置总是在 src/main/java 中，配置文件则是在 src/main/resources 中，编译好的类总是放在项目的 target 目录下，总的来说，Maven 实现了以下目标：

- 使构建项目变得很容易，Maven 屏蔽了构建的复杂过程。比如，你只需要输入 maven package 就可以构建整个 Java 项目。
- 统一了构建项目的方式，不同人、不同公司的项目都有同样的描述项目和构建项目的方式，Maven 通过 pom.xml 来描述项目，并提供一系列插件来构建项目。只要熟悉了 Maven，就熟悉了所有项目的构建方式。
- 提出了一套开发项目的最佳实践，而不用每个项目都有不同结构和构建方式，比如源代码在 src/main/java 中，测试代码则在 src/test/java 中，项目需要的配置文件则放在 src/main/resources 中。
- 包含不同环境项目的构建方式。
- 解决了类库依赖的问题，只需要声明使用的类库，Maven 会自动从仓库下载依赖的 jar 包，并能协助你管理 jar 包之间的冲突。

> **注意：**仓库有多种，位于 Apache 的是中心仓库，是 jar 发布存放的地方，可以通过 http://search.maven.org/查询仓库和下载 jar 文件。国内外也有大量镜像库，与中心仓库同步，比如国内的阿里云提供的 Maven 镜像库。也可以用 Sonatype 创建一个私服，用来发布和存放库以提高下载速度或者存放公司私有的 jar。
>
> 无论依赖的 jar 包来自哪个仓库，在开发人员本地，Maven 都会创建一个本地仓库来缓存已经下载的 jar 包，以避免每次都去仓库下载。这个本地仓库位默认在用户目录下的.m2 隐藏文件夹中。

Maven 的核心是 pom.xml，用 XML 方式描述了项目模型，pom 通常有以下元素。

- groupId：表示项目所属的组，通常是一个公司或者组织的名称，如 org.springframework。
- artifactId：项目唯一的标识，比如，有 spring-boot-starter-web、spring-boot-devtools、

groupId 和 artifactId 能唯一标识一个项目或者一个库，我们通常称之为项目坐标。

- packaging：项目的类型，常用的有 jar 和 war 两种，jar 表示项目会打包成一个 jar 包，这是 Spring Boot 的默认使用方式。
- version：项目的版本号，比如 0.0.1-SNAPSHOT、1.5.2.RELEASE。

> 通常来说，项目版本号分三段，主版本号.次版本号.修订版本号。主版本号变动代表架构变动或者不兼容实现，次版本号是兼容性修改、功能增强，修订版本号则是 bug 修复。
>
> 版本的后缀意味着项目的不同阶段，SNAPSHOT 表示开发中的版本，会修复 bug 和添加新功能；RELEASE 表示的是一个正式发布版，中间还可能有 M1、M2(M 指里程碑，即将发布)、RC(Release Candidate，发布候选)、GA(general availability，基本可用版本)等表示即将发布前的各个过程，SNAPSHOT<m1<m2…<RC<GA<Realease。

- modelVersion：代表 pom 文件的 Maven 的版本，如本书写作的时候 Maven 的 modelVersion 是 4.0.0。
- dependencies：此元素下包含了多个 dependency，用来声明项目的依赖，这是 pom 最核心的部分。
- dependency：包含在 dependencies 中，用来声明项目的依赖，比如项目用到的 MySQL 驱动。

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.5</version>
    <scope>provided</scope>
</dependency>
```

依赖的坐标是 mysql/mysql-connector-java，版本号是 6.0.5。

- scope：scope 代表此类库与项目的关系，默认是 compile，也就是编译和打包都需要此类库。test 表示仅仅在单元测试的时候需要；provided 表示在编译阶段需要此类库，但打包阶段不需要，这是因为项目的目标环境已经提供了；runtime 表示在编译和打包的时候都不需要，但在运行的时候需要，比如某个指定的数据库驱动，编译和打包都不需要，但测试应用要连到数据库时就需要此数据库驱动。
- build：此项在 pom 中可选，build 包含了多个插件 plugin，用来辅助项目构建。Maven

与以往的 Ant 等其他构建工具不同，Maven 已经约定俗成地包含了构建方法，插件可以在构建过程中影响项目的构建。

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
  </plugin>
</build>
```

Maven 默认的 Java 源码编译方式根据平台的不同或者 Maven 本身版本的不同有所差别，如早期 Maven 版本默认编译的是 Java1.6，因此，这个插件声明了使用 Java1.8 的编译方式，源码使用的 encoding 是 UTF-8。

由于篇幅有限，将不作关于 Maven 的更多介绍，对于 Spring Boot 初学者来说，Spring Boot 已经将 Maven 的使用更加简单化了，正如第 1 章看到的，编写一个 Spring Boot 的 Web 应用，项目采用的 Maven 配置是非常简单的，所以本书所涉及的 Maven 知识非常少。

## 2.2.2 安装 Maven

我们在前面安装 Eclipse 的时候，Eclipse 就已经自带了一个 Maven 版本，但此版本较低，我们需要安装一个新的 Maven 版本。

- 从 maven.apache.org 上下载最新的 Maven。
- 直接解压，比如 D:\apache\maven-3.3.9。
- 设置环境变量，Windows 系统类似如下：

```
MAVEN_HOME = d:\apache\maven-3.3.9
path = %MAVEN_HOME%\bin;D:\Java\jdk1.8.0_73\bin;...
```

- 进入命令行，验证 Maven 是否安装成功，输入 mvn –version，应该有如下输出（以笔者的 Mac 为例）：

```
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06;
2015-04-22T19:57:37+08:00)
Maven home: /Users/xiandafu/apache/apache-maven-3.3.3
Java version: 1.8.0_45, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/
Home/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.10.5", arch: "x86_64", family: "mac"
```

如果没有显示上述信息，请确认 JDK 环境配置是否正确，Maven 是否按照上面的步骤配置。

## 2.2.3 设置 Maven

使用 Maven 的唯一问题可能就是网络问题，Maven 管理了依赖库，默认情况下都是从中心仓库下载。中心仓库位于国外，下载速度非常慢，因此用 Maven 构建项目的时候，经常卡在下载超时的过程中。幸好国内有几个镜像网站可以用，因此在配置 Maven 时，应优先使用国内镜像网站，当国内镜像仓库下载失败（因为有可能没有需要的依赖库）的时候，再切换到中心仓库下载。

- 进入 Maven 的安装目录，进入 conf 目录，编辑 settings.xml。
- 找到 mirrors 元素，添加如下仓库镜像：

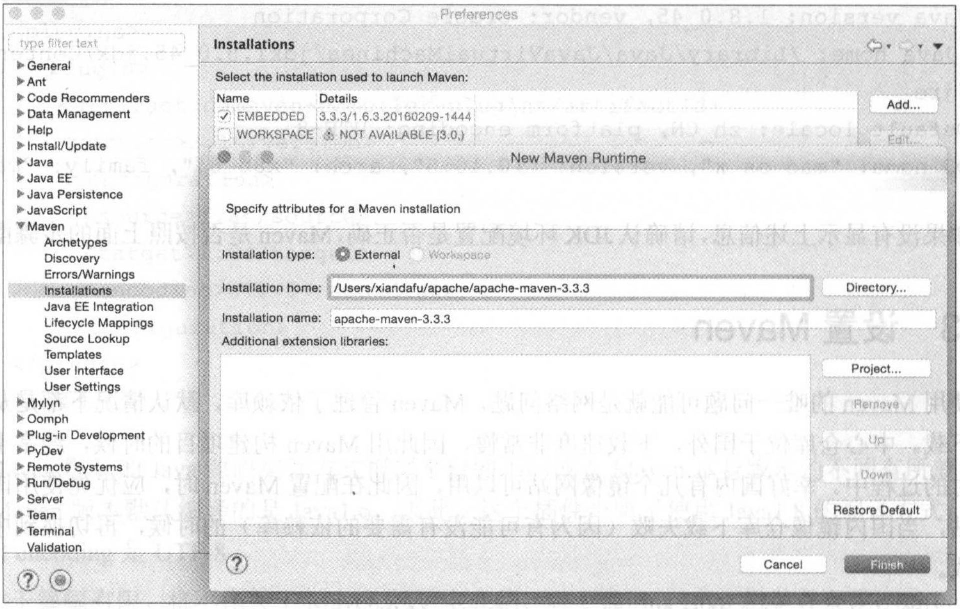
```
<mirror>
  <id>alimaven</id>
  <name>aliyun maven</name>
<url>http://maven.aliyun.com/nexus/content/groups/public/</url>
  <mirrorOf>central</mirrorOf>
</mirror>
```

> **注意**：镜像不一定包含所有的依赖库，如果 Maven 在构建项目中报出缺少依赖库，请暂时注释掉这个镜像并重新构建，切换到中心仓库。

## 2.2.4 使用 IDE 设置 Maven

安装配置好 Maven 后，需要在 Eclipse 中配置，使用你下载配置好的 Maven 而不是 Eclipse 自带的 Maven，打开 Eclipse 的偏好设置（Window|Preference 或者 Mac 的 Eclipse |Prefrence）。

- 点击左侧导航栏，找到 Maven。
- 点击 Maven 下的 Installations。
- 如下图所示，可以看到默认安装了 3.3.3 版本。



- 选择面板右侧，点击“Add”，在弹出的面板里，在“Installation home”中输入 Maven 的安装目录，然后点击确定按钮。
- 回到主面板，勾选使用新的 Maven，配置成功。

### 2.2.5 Maven 的常用命令

在上一章中，Eclipse 已经集成了最新的 Maven，本书后面将介绍如何通过 Maven 管理和构建 Spring Boot 应用。这里简单介绍一下 Maven 常用的构建命令，以方便你理解 Maven 的使用方法。

首先创建一个目录，比如 test 目录，我们在此目录下创建 Maven 的 Java 工程。进入此目录创建一个 Maven 工程：

```
mvn -B archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app
```

如上所示，指定了项目坐标，groupId 是 com.mycompany.app，项目的标识是 my-app，archetypeGroupId 是生成 Maven 项目的模板，Maven 提供了很多模板用来生成不同应用的 Maven 项目。

第一次运行需要花较长的时间，主要是从阿里云上下载 Maven 本身需要的类库，因为我们前一章安装的是 Maven 的核心库，为了用 Maven 构建项目，Maven 还需要下载其他类库和插件，在下载完需要的插件后构建项目，大概需要数十秒时间，直到看到如下内容：

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.085 s
```

这时检查 test 目录，Maven 已经创建了一个项目工程 my-app，里面有一系列文件，最主要的就是 pom.xml，内容如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

mvn compile：编译 Maven 工程，进入 my-app，运行 mvn compile，能看到如下提示。

```
[INFO] Compiling 1 source file to /Users/xiandafu/apache/test/my-app/
target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.778 s
```

mvn package：编译并打包工程，根据 pom.xml 中元素 packaging 是 jar 还是 war 进行打包，本例中，会在 target 目录下生成一个 jar 包。

```
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ my-app ---
[INFO] Building jar: /Users/xiandafu/apache/test/my-app/target/my-app-
1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.405 s
```

mvn install：打包并安装到本地仓库。如果你的项目是一个基础类库，本地其他项目也需要，则需要安装到本地仓库。这样，其他本地 Maven 项目就可以通过项目坐标引用。mvn install 有类似如下输出：

```
[INFO] Building jar: /Users/xiandafu/temp/test/my-app/target/my-app-1.0-
SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ my-app ---
[INFO] Installing /Users/xiandafu/temp/test/my-app/target/my-app-1.0-
SNAPSHOT.jar to /Users/xiandafu/.m2/repository/com/mycompany/app/my-app/1.0-
SNAPSHOT/my-app-1.0-SNAPSHOT.jar
[INFO] Installing /Users/xiandafu/temp/test/my-app/pom.xml to /Users/
xiandafu/.m2/repository/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNA
PSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

本地仓库默认位于用户目录的.m2 目录下。

mvn deploy：同 install，但打包并安装到远程仓库。本书不会用到这个命令。

mvn clean：删除 target 目录。

> Maven 的仓库有两大类，第一类是远程仓库，包括中心仓库，位于 http://search.maven.org/；还包括镜像仓库，比如国内常用的镜像 http://maven.aliyun.com/nexus/，还有利用 nexus 软件自己搭建的公司私服。
>
> 还有一类是本地仓库，位于用户目录的.m2 目录下，远程仓库加载的库总会先放到本地仓库作为缓存。mvn install 也会将项目打包安装到本地仓库。

## 2.3 Spring 核心技术

本书直接从 Spring Boot 入手，并不要求读者具备 Spring 的知识，但是了解一下 Spring 的历史和特性，还是有助于 Spring boot 项目开发的。

### 2.3.1 Spring 的历史

Java EE 是企业应用需求的体现，而 Spring 则基于企业应用并非全是分布式这个前提，进一步简化了 Java EE 的开发，Spring 奠基作者 Rod Johnson 在 2003 年出版的《*Expert One-On-One Java EE Design and Development*》一书中首次提到 Spring 框架，它并不像当时的 Java EE 一样给了开发人员诸多限制。

2004 年，Spring 又推出了 IoC（反向控制）和 AOP（面向切面编程），使得 Spring 成为非常受欢迎的框架，开发企业应用的另外一种轻量级选择。

2005 年，Spring 成立了独立的公司，公司最早的名字是 interface21，现在经过一系列并购，称为 Pivotal，全心全意维护 Spring 框架，同时，大量的社区人员参与进来，商业公司也参与了 Spring 的开发。

> Pivotal 是 EMC、VMware、GE 共同成立的公司，目前估值为 28 亿美元，Spring 的开发人员基本上都是这个公司雇佣人员，一些知名的开源产品的顶级开发者也成为该公司的雇佣人员。
>
> 知名的 Redis、RabbitMQ 也属于这个公司。

如今 Spring 框架已经发展成为一个庞大的开源帝国，包含了多款开源软件，主要有：

- Spring Framework；
- Spring Boot；
- Spring Data 系列；
- Spring Cloud 系列。

还有很多未在这里一一列出，有兴趣的读者可以进一步参考 Spring 文档，本书涉及 Spring Boot、Spring Framework、Spring Data 等知识。

> Spring Cloud 是现在炙手可热的技术，是一个基于 Spring Boot 实现的云应用开发工具，它为基于 JVM 的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。可以说 Spring Boot 和 Spring Cloud 是一对黄金搭档，也是Spring团队重点的开发方向。市场上实现的微服务系统架构，主要是联合这两种技术，笔者希望通过本书为读者实现微服务架构打下基础，这也是笔者的初衷之一。

## 2.3.2 Spring 容器介绍

### 2.3.2.1 Spring IoC

Spring 框架的实现依赖 IoC（Inversion of Control，反向控制）原则，更为形象的称呼是 DI（dependency injection，依赖注入）。相对于 Bean 直接构造依赖的对象，Spring 框架则根据 Bean 之间的依赖关系创建对象，并注入到 Bean 中。对于传统的企业应用代码，以下是一种常见的写法：

```
public UserServiceImpl implements UserService {
  private CreditUserService  creditUserService = new CreditUserService();
  public void order(....){
    ....
    creditUserService.addCredit(5);
  }
}
```

对于上述代码，creditUserService 实例是在 UserService 对象中创建的，创建方式比较简单，调用了构造函数。如果 creditUserService 本身构造方式比较复杂，而且是个单例对象，那么会使用工厂类来实现进一步优化，比如：

```
private CreditUserService creditUserService = CreditUserServiceFactory.
getService();
```

或者更先进一点的做法：

```
private CreditUserService creditUserService = ServiceFactory.getService
(CreditUserService.class);
```

ServiceFactory 是一个工厂类，根据传入的类名，返回一个已经初始化好的实例。

在 Spring 框架中，如果你的 Bean 是通过 Spring 来管理的，Spring 容器则会帮你完成这些事情，开发人员需要做的仅仅是声明依赖关系，因此代码如下所示。

```
@Service
public CreditUserServiceImpl implements CreditUserService {
  public void addCredit(int score){

  }
}

@Service
public class UserService {
  @Autowired
  private CreditUserService  creditUserService ;
  public void order(....){
    ...
    creditUserService.addCredit(5);
  }
}
```

Bean 通过注解@Service 声明为一个 Spring 管理的 Bean，Spring 容器会扫描 classpath 下的所有类，找到带有@Service 注解的 UserService 类，并根据 Spring 注解对其进行初始化和增强，如果发现此类属性 creditUserService 也有注解@Autowired，则会从 Spring 容器里查找一个已经初始化好的 CreditUserService，如果没有，则先初始化 CreditUserService。

Spring 的容器构造和管理 Bean 远远比这个例子复杂，但作为 Spring Boot 应用开发者，不需要了解这么深入，Spring 容器把简单留给了开发者，作为一个 Spring 新手，只需要对这些概念有所了解即可。

> 如何成功吸引 Spring 容器的注意而“有幸”成为容器管理的 Bean？在 Spring Boot 中就是依靠注解，如在类上的注解@Controller、@Service、@Configuration 等，方法上的注解 @Bean。
>
> 还有一种吸引 Spring 容器注意的办法是实现 Spring 的某些接口类，Spring 会在容器的生命周期里调用这些接口类，比如 BeanPostProcessor 接口就是在所有容器管理 Bean 初始化后，会调用此接口实现，对 Bean 进行进一步的配置，比如 Spring 的 AutowiredAnnotationBeanPostProcessor 会寻找 Bean 里的@Autowired 注解来注入依赖 Bean。

#### 2.3.2.2 Spring 常用注解

Spring 提供了多个注解声明 Bean 为 Spring 管理的 Bean，注解不同代表的含义不一样，但对于 Spring 容器来说，都是 Spring 管理的 Bean。

- Controller：声明此类是一个 MVC 类，通常与@RequestMapping 一起使用。

```
@COntroller
@RequestMapping("/user")
public class UserController {
  @RequestMapping("/get/{id}")
  public String getUser(@PathVariable String id){

  }
}
```

如用户访问的地址是/user/get/1，将调用 getUser 方法，并把参数 1 传给 id。

- Service：声明此类是一个业务处理类，通常与@Transactional 一起配合使用。

```
@Service
@Transactional
public Class UserServiceImpl implements UserService{
  public void order(..){
  }
}
```

- Repository：声明此类是一个数据库或者其他 NoSQL 访问类。

```
@Repository
public class UserDao implements CrudDao<User,String> {

}
```

- RestController：同 Controller，用于 REST 服务。
- Component：声明此类是一个 Spring 管理的类，通常用于无法用上述注解描述的 Spring 管理类。
- Configuration：声明此类是一个配置类，通常与注解@Bean 配合使用。

```
@Configuration
public class DataSourceConfig {
    @Bean(name = "dataSource")
    public DataSource datasource(Environment env) {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(env.getProperty("spring.datasource.url"));
        ds.setUsername(env.getProperty("spring.datasource.username"));
        ds.setPassword(env.getProperty("spring.datasource.password"));
        ds.setDriverClassName(env.getProperty("spring.datasource.driver-
class-name"));
        return ds;
    }
}
```

如上示例，DataSourceConfig 是一个 Spring 容器配置类，配置了 HikariDataSource。

- Bean：作用在方法上，声明该方法执行的返回结果是一个 Spring 容器管理的 Bean，参考@Configuration 示例。

Spring 负责实例化 Bean，开发者可以提供一系列回调函数，用于进一步配置 Bean，包括@PostConstruct 注解和@PreDestory 注解。

当 Bean 被容器初始化后，会调用@PostConstruct 的注解方法：

```
@Component
public class ExampleBean  {
  @PostConstruct
  public void init() {
```

```
  }
}
```

@PreDestory：在容器被销毁之前，调用被@PreDestory 注解的方法。

```
@Service
public class ExampleBean {

  @PreDestory
  public void cleanup() {
  }
}
```

> 由于传统原因，Spring 还提供了其他 Bean 声明周期的回调方式，指定初始化和销毁的方法，以及可以用实现 InitializingBean 接口的 afterPropertiesSet()来初始化 Bean 和实现 DisposableBean 的 destroy()方法来销毁 Bean。

Spring 有两种方式来引用容器管理的 Bean，一种是根据名字，为每个管理的 Bean 指定一个名字，随后可以通过名字引用此 Bean。

```
@Service()
@Qualifier("exampleBean")
public class ExampleBean {

}
```

这样在其他 Bean 中，可以使用注解@Qualifier 来引用，比如：

```
@Service
public AnotherExampleBean{
@Qualifier("exampleBean") ExampleBean bean;
}
```

另外一种是根据类型，使用起来更加简单。上面的例子可以改写成：

```
@Service
public class ExampleBean  {
```

```
}

@Service
public AnotherExampleBean{
  @Autowried ExampleBean bean;

}
```

在一个 Spring 管理的 Bean 中，开发人员可以通过@Autowired 声明对其他 Bean 的引用，作用于属性或者构造函数参数，甚至是方法调用参数上。

## 2.3.3 Spring AOP 介绍

### 2.3.3.1 AOP 介绍

AOP（Aspect-Oriented Programming，面向切面编程）提供了另外一种思路来实现应用系统的公共服务。

- 每个业务方法调用的权限管理：在用户调用方法前判断用户是否有权调用此方法。我们在第 1 章曾有一个这样的简单示例。
- 每个方法调用的审计：记录谁调用了什么业务方法，传入参数是什么，操作是否成功。
- 数据库事务的管理：在执行数据库一系列操作前，先开启事务，在执行完后提交事务；如果执行出错，则回滚事务。
- 缓存：对业务方法返回的数据进行缓存，下次调用的时候，如果参数未变，直接从缓存中获取数据，而不再调取应用方法。

> 在阅读下面关于 AOP 的技术细节后，也许因为看不懂而有点沮丧，不过没关系，没有人能看一次 AOP 知识点就能了解 AOP。Spring AOP 实现起来非常简单，我们将在下一节继续描述如何在 Spring 中如何实现 AOP。

在没有 AOP 之前，这些应用需求都需要程序员实现在上面所述的每个方法中。有了 AOP，应用可以在运行时刻动态地在方法调用前后“织入”一些公共代码，从而提供系统的公共服务。AOP 有如下术语。

- Aspect：Aspect 声明类似于 Java 中的类声明，在 Aspect 中会包含一些 Pointcut 及相应的 Advice。

- Joint point：表示在程序中明确定义的点，典型的包括方法调用、对类成员的访问，以及异常处理程序块的执行等。Spring 中的 Joint point 只支持方法调用。
- Pointcut：表示一组 Joint point，如方法名、参数类型、返回类型等，这些 Joint point 通过逻辑关系组合起来，它定义了相应的 Advice 将要发生的地方。简单理解 Pointcut（一种表达式）——用来判断在 Joint point（方法调用）中执行 Advice（操作）。
- Advice：Advice 定义了在 Pointcut 里面定义的程序点具体要做的操作，它通过 before、around、after（return、throw、finally）来区别是在每个 Joint point 之前、之后还是执行前后要调用的代码。
  - before：在执行方法前调用 Advice，比如 cache 功能可以在执行方法前先判断是否有缓存。
  - around：在执行方法前后调用 Advice，这是 Spring 框架和应用系统一种最为常用的方法，本书第 1 章用户权限的实现就是采用的 around。
  - after：在执行方法后调用 Advice，after return 是方法正常返回后调用，after throw 是方法抛出异常后调用。
  - finally：方法调用后执行 Advice，无论是否抛出异常还是正常返回。
- AOP proxy：AOP Proxy 也是 Java 对象，由 AOP 框架创建，用来完成上述的动作，AOP 对象通常可以通过 JDK dynamic proxy 完成，或者使用 CGLib 完成。
- Weaving：实现上述切面编程的代码织入，可以在编译时刻（通过 AspectJ compiler），也可以在运行时刻，Spring 和其他大多数 Java 框架都是在运行时刻生成代理。

#### 2.3.3.2 在 Spring Boot 中使用 AOP

首先，参考第 1 章，创建一个简单的 Spring Boot 应用。

其次，引入 AOP 依赖，在 pom 文件中添加 spring-boot-starter-aop：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

编写一个 AOP 切面类：

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@Aspect
public class AOPConfig {
  @Around("@within(org.springframework.stereotype.Controller) ")
  public Object simpleAop(final ProceedingJoinPoint pjp)  throws Throwable {
    try {
      Object[] args = pjp.getArgs();
      System.out.println("args:"+Arrays.asList(args));
      // 调用原有的方法
      Object o = pjp.proceed();
      System.out.println("return :"+o);
      return o;

    } catch (Throwable e) {
      throw e;
    }
  }
}
```

对于这段代码，可以按照以下顺序理解：

（1）@Configuration，用于声明这是一个 Spring 管理配置 Bean，这里是引起 Spring 的注意，好比一群人里有一个人带了一个很高的帽子，更能吸引人注意。

（2）@Aspect，声明了这是一个切面类。

（3）@Around，声明了一个表达式，描述要织入的目标的特性，比如@within 表示目标类型带有注解，其注解类型参数为 org.springframework.stereotype.Controller，这意味着 Spring Controller 方法在被调用的时候，都会执行@Around 注解的方法，也就是 simpleAop。

（4）simpleAop 是用来织入的代码，其参数称为 ProceedingJoinPoint，如上述代码实例，将调用的方法的参数取出来，打印到控制台。

（5）pjp.proceed()，通常情况下，执行完切面代码，还需要继续执行应用代码，proceed()方法则会继续调用原有的业务逻辑，并将返回对象正常返回。

继续执行应用代码，有可能抛出异常，在切面里，我们不会处理这个异常，直接抛出给调用者。

当我们访问 http://127.0.0.1:8080/sayhello.html?name=a 的时候，查看控制台，会发现有如下输出：

```
args:[a]
return :hello world
```

说明织入的代码被执行了。

> Spring AOP 支持多种表达式及表达式的组合，这里列出一些简单的表达式以供参考，更多的表达式需要参考 Spring 官网文档。
>
> - execution(public * *(..))：所有 public 方法，后面的星号代表类路径和方法名。
> - execution(* set*(..))：所有 set 开头的方法。
> - execution(public set*(..))：所有 set 开头的 public 方法。
> - execution(public com.xyz.service.* set*(..))：所有 set 开头的 public 方法，且位于 com.xyz.service 包下。
> - target(com.xyz.service.CommonService)：所有实现了 CommonService 接口的类的方法。
> - @target(org.springframework.transaction.annotation.Transactional)：所有用 @Transactional 注解的方法。
> - @within(org.springframework.stereotype.Controller)：类型声明了@Controller 的所有方法。

# 第 3 章 MVC 框架

在 Spring 框架和 Spring Boot 中，最常用的技术就是 MVC 框架。试图讲清楚 Spring MVC 的内容，有可能需要一本书来讲述。本章将介绍 MVC 中最实用的部分，一些不常用的技术，或者过时的技术将不做介绍。

MVC 框架会处理类似如下相同的技术需求：

- HTTP URL 映射到 Controller 某个方法；
- HTTP 参数映射到 Controller 方法的参数上，比如参数映射到某个 Java 对象，或者上传附件映射到某个 File 对象上；
- 参数的校验；
- MVC 错误处理；
- MVC 中如何调用视图；
- MVC 中如何序列化对象成 JSON；
- 拦截器等高级定制。

Spring MVC 具备上述所有技术实现，而且可能是现在 Java 开源框架中功能最全的 MVC 框架，本章将依次介绍 Spring MVC 是如何提供这些技术需求的。

# 3.1 集成 MVC 框架

## 3.1.1 引入依赖

Spring Boot 集成 Spring MVC 框架并实现自动配置，只需要在 pom 中添加以下依赖即可，不需要其他任何配置：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

本章中的例子使用了 Beetl 作为模板技术，因此还需要添加以下依赖：

```
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl-framework-starter</artifactId>
  <version>1.1.15.RELEASE</version>
</dependency>
```

## 3.1.2 Web 应用目录结构

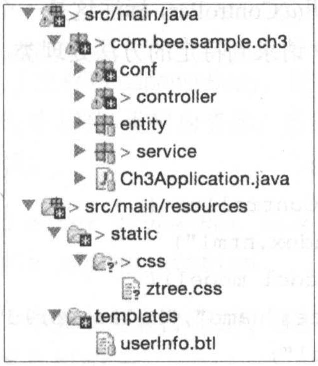
Web 的模板文件位于 resources/templates 目录下，模板文件使用的静态资源文件，如 JS、CSS、图片，存放在 resources/static 目录下。在 MVC 中，视图名自动在 templates 目录下找到对应的模板名称，模板中使用的静态资源将在 static 目录下查找。如下 Controller 代码所示，将“/userdetail.html?id=xxx”请求映射到 foo 方法：

```
@RequestMapping("/userdetail.html")
public  String foo(String id){
  .....
  return "/admin/userInfo.btl;
}
```

渲染的视图名称是“/admin/userInfo.btl”，会寻找 templates/admin/userInfo.btl 模板文件，如果这个 userInfo.btl 有以下引用：

```
<link href="/css/ztree.css" rel="stylesheet"/>
```

则 Spring Boot 会从 static 下查找 css/ztree.css，整个目录结果看起来类似下图：




## 3.1.3 Java 包名结构

从上图可以看到，在 Spring Boot 应用中，通常会创建如下子包名：

- Controller——此包下包含了 MVC 的 Controller，如 UserController；
- Service——此包下有业务处理代码，如 UserService；
- entity——包含了业务实体，如 User 类；
- conf——包含了一些配置类，比如用于配置数据源的 DataSourceConfig，还有本章的 JSON 序列化配置 JacksonConf。

Spring Boot 应用的程序入口 Ch3Application 建立在这些包名上，这样 Spring Boot 应用能自动扫描整个项目的结构。

Ch3Application 同其他 Spring Boot 程序一样，仅仅是一个带有@SpringBootApplication 注解的类：

```
@SpringBootApplication
public class Ch3Application {
    public static void main(String[] args) {
        SpringApplication.run(Ch3Application.class, args);
    }
}
```

## 3.2 使用 Controller

Spring MVC 框架不像传统的 MVC 框架那样必须继承某个基础类才能处理用户的 HTTP 请求，Spring MVC 只需要在类上声明@Controller，标注这是一个 Controller 即可。对于用户请求，使用@RequestMapping 映射 HTTP 请求到特定的方法处理类。

```
@Controller
@RequestMapping("/test")
public class HelloworldController {
  @RequestMapping("/index.html")
  public  String say(Model model){
    model.addAttribute("name","hello,world");
    return "/index.btl";
  }
}
```

如以上代码所示，@Controller 作用于类上，表示这是一个 MVC 中的 Controller。

@RequestMapping 既可以作用在方法上，也可以作用在类上。如上例所示，用户如果访问/test/index.html，则会交给 HelloworldController.say 方法来处理。

say 方法有一个参数 Model，这是 Spring MVC 可识别的一个参数类型（将在 3.4 节详细讲述方法参数），用来表示 MVC 中的 Model。你可以在 say 方法中为这个 Model 添加多个变量，这些变量随后可以用于视图渲染。

say 方法返回的类型是字符串，默认是视图的名称。Spring Boot 的视图默认保存在 resources/templates 目录下，因此，渲染的视图是/resources/templates/index.btl 模板文件。

MVC 框架有时候返回的是 JSON 字符串，如果想直接返回内容而不是视图名，则需要在方法上使用@ResponseBody：

```
@RequestMapping("/index.json")
public @ResponseBody String say(){
  return  "hello,world";
}
```

ResponseBody 注解直接将返回的对象输出到客户端，如果是字符串，则直接返回；如果不是，则默认使用 Jackson 序列化成 JSON 字符串后输出。

```
@RequestMapping(path="/all.json" ,method = RequestMethod.GET)
public  @ResponseBody List<User> allUser() {
  return userService.allUser();
}
```

> 如果你期望返回 JSON，使用了注解@ResponseBody，但你的请求 URL 以 html 结尾，这会导致 Spring Boot 认为请求的是 HTML 类型的资源，而返回类型是 JSON 类型资源，与期望类型不一致而报出如下错误：
>
> ```
> There was an unexpected error (type=Not Acceptable, status=406).
> Could not find acceptable representation
> ```
>
> 建议在 Spring Boot 应用中，如果期望返回 JSON，URL 请求资源后缀是 json；如果期望返回视图，URL 请求资源后缀是 html。

> 在开发过程中，测试 Controller 最简单的方法是通过浏览器直接访问地址，然而对于 POST 请求、文件上传，还有更多的 HTTP 请求控制，打开浏览器直接访问的方式不可行，因此 3.10 节介绍了 curl 命令，可以用于测试你写好的 Controller 方法。你也可以用 Spring Test 来测试 Controller 方法。
>
> 建议熟悉 curl 命令后再来测试 Controller，可以提高开发效率。

## 3.3 URL 映射到方法

从本节开始，将学习 HTTP 请求到 Spring Controller 的方法的映射，Spring MVC 提供了各种各样的映射方式，尽管会让 Spring MVC 框架的学习成本提高，但 Spring MVC 框架能让 Web 应用的代码更容易阅读和维护。

建议学习本章知识后，可以按照前面提到的“故意制造一些错误”的学习方法，看看 Spring MVC 有什么错误提示，这样有助于加深对 Spring MVC 的理解。

### 3.3.1 @RequestMapping

你可以使用@RequestMapping 来映射 URL，比如/test 到某个 Controller 类，或者是某个具体的方法。通常类上的注解@RequestMapping 用来标注请求的路径，方法上的@RequestMapping

注解进一步映射特定的 URL 到某个具体的处理方法。

RequestMapping 有多个属性来进一步匹配 HTTP 请求到 Controller 方法，分别是：

- value，请求的 URL 的路径，支持 URL 模板、正则表达式。
- method，HTTP 请求方法，有 GET、POST、PUT 等。
- consumes，允许的媒体类型（Media Types），如 consumes = "application/json"，对应于请求的 HTTP 的 Content-Type。
- produces，相应的媒体类型，如 produces="application/json"，对应于 HTTP 的 Accept 字段。
- params，请求的参数，如 params="action=update"。
- headers，请求的 HTTP 头的值，如 headers = "myHeader=myValue"。

## 3.3.2 URL 路径匹配

属性 value 用于匹配一个 URL 映射，value 支持简单的表达式来匹配：

```
@RequestMapping(value="/get/{id}.json")
public  @ResponseBody  User getById( @PathVariable("id") Long id) {
  return userService.getUserById(id);
}
```

如上面的例子所示，访问路径是/get/1.json，将调用 getById 方法，且参数 id 的值是 1。注解 PathVariable 作用在方法参数上，用来表示参数的值来自于 URL 路径。

如果你的 IDE 环境启用了 debug 模式（通常 IDE 或者 Maven 都会启用），则在编译 Java 代码成为字节码的时候，方法的参数保留了原来的参数名字。Java8 如果使用了 parameters 编译选项，也会保留参数名字。Spring 可以在这种情况下识别 URL 中存在的表达式与方法参数的对应关系，从而自动赋值，因此上述代码通常可以简化成：

```
@RequestMapping(path="/user/{id}.json" ,method = RequestMethod.GET)
public  @ResponseBody User getById( @PathVariable Long id) {
  return userService.getUserById(id);
}
```

也可以使用类似 Ant 的通配符来对 URL 进行映射，比如：

```
@RequestMapping(path="/user/all/*.json" ,method = RequestMethod.GET)
@ResponseBody
public  List<User> allUser() {
 return userService.allUser();
}
```

> **Ant 路径表达式**
>
> Ant 用符号“*”来表示匹配任意字符，用“**”来表示统配任意路径，用“?”来匹配单个字符，比如：
>
> - /user/*.html，匹配/user/1.html、/user/2.html 等。
> - /**/1.html，匹配/1.html，也匹配/user/1.html，还匹配/user/add/1.html。
> - /user/?.html，匹配/user/1.html，但不匹配/user/11.html。

如果一个请求有多个@RequestMapping 能够匹配，通常是更具体的匹配会作为处理此请求的方法。

- 有通配符的低于没有通配符的，比如/user/add.json 比/user/*.json 优先匹配；
- 有“**”通配符的低于有“*”通配符的。

URL 映射也可以使用${}来获得系统的配置或者环境变量，通常用于 Controller 路径是通过配置文件设定的情况。

```
@RequestMapping("/${query.all}.json" )
@ResponseBody
public  List<User> all() {
 return userService.allUser();
}
```

## 3.3.3 HTTP method 匹配

@RequestMapping 提供 method 属性来映射对应 HTTP 的请求方法，通常 HTTP 请求方法有如下内容：

- GET，用来获取 URL 对应的内容。
- POST，用来向服务器提交信息。
- HEAD，同 GET，但不返回消息体，通常用于返回 URL 对应的元信息，如过期时间等。搜索引擎通常用 HEAD 来获取网页信息。

- PUT，同 POST，用来向服务器提交信息，但语义上更像一个更新操作。同一个数据，多次 PUT 操作，也不会导致数据发生改变。而 POST 在语义上更类似新增操作。
- DELETE，删除对应的资源信息。
- PATCH 方法，类似 PUT 方法，表示信息的局部更新。

通常对于 Web 应用，GET 和 POST 是经常使用的选项，对于 REST 接口，则会使用 PUT、DELETE 等用来从语义上进一步区分操作。

Spring 提供了简化后的@RequestMapping，提供了新的注解来表示 HTTP 方法：

- @GetMapping；
- @PostMapping；
- @PutMapping；
- @DeleteMapping；
- @PatchMapping。

因此，上一章的例子可以修改为：

```
@GetMapping("/user/all/*.json")
public  @ResponseBody List<User> allUser() {
  return userService.allUser();
}
```

## 3.3.4 consumes 和 produces

属性 consumes 意味着请求的 HTTP 头的 Content-Type 媒体类型与 consumes 的值匹配，才能调用此方法。

```
@GetMapping(value="/consumes/test.json",consumes = "application/json" )
@ResponseBody
public  User forJson() {
  return userService.getUserById(1l);
}
```

这里映射指定请求的媒体类型是 application/json，因此，此方法接受一个 AJAX 请求。如果通过浏览器直接访问，则会看到 Spring Boot 报出如下错误，因为通过浏览器访问，通常并没有设置 Content-Type，所以说 null 不支持。

```
There was an unexpected error (type=Unsupported Media Type, status=415).
Content type 'null' not supported
```

为了成功调用上述 Controller 方法，AJAX 调用必须设置 Content-Type 为 application/json，如以下 JS 代码所示。

```
$.ajax({
  type: "get",
  url: "/consumes/test.json",
  contentType: "application/json",
  ....
});
```

produces 属性对应于 HTTP 请求的 Accept 字段，只有匹配得上的方法才能被调用。

```
@GetMapping(path = "/user/{userId}", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
@ResponseBody
public User getUser(@PathVariable Long userId, Model model) {
  return userService.getUserById(userId);
}
```

通常浏览器都会将 Accept 设置为*.* ，因此通过浏览器直接访问“/user/1”，浏览器总是返回 id 为 1 的用户信息，并转成 JSON 格式。

## 3.3.5 params 和 header 匹配

可以从请求参数或者 HTTP 头中提取值来进一步确定调用的方法，有以下三种形式：

- 如果存在参数，则通过；
- 如果不存在参数，则通过；
- 如果参数等于某一个具体值，则通过。

```
@PostMapping(path = "/update.json", params = "action=save")
@ResponseBody
public void saveUser() {
  System.out.println("call save");
}
```

```
@PostMapping(path = "/update.json", params = "action=update")
@ResponseBody
public void updateUser() {
  System.out.println("call update");
}
```

header 也与 params 一样：

```
@GetMapping(path = "/update.json", headers = "action=update")
@ResponseBody
public void updateUser() {
   System.out.println("call update");
}
```

> 很多 MVC 框架并未提供如此多的匹配方式来定位到调用方法，如果你了解 Servlet 规范，可能会认为可以在方法中通过 HttpServletRequest 获取到请求 URL、HTTP 头等信息后再做进一步处理。Spring 在方法签名处提供了多种匹配方式，能进一步规范方法调用，提高代码的可阅读性，从而更容易维护 Spring Boot 应用代码。

## 3.4 方法参数

Spring 的 Controller 方法可以接受多种类型参数，比如我们看到的 path 变量，还有 MVC 的 Model。除此之外，方法还能接受以下参数。

- @PathVariable，可以将 URL 中的值映射到方法参数中。
- Model，Spring 中通用的 MVC 模型，也可以使用 Map 和 ModelMap 作为渲染视图的模型。
- ModelAndView，包含了模型和视图路径的对象。
- JavaBean，将 HTTP 参数映射到 JavaBean 对象。
- MultipartFile，用于处理文件上传。
- @ModelAttribute，使用该注解的变量将作为 Model 的一个属性。
- WebRequest 或者 NativeWebRequest，类似 Servlet Request，但做了一定封装。
- java.io.InputStream 和 java.io.Reader，用来获取 Servlet API 中的 InputStream/Reader。
- java.io.OutputStream / java.io.Writer，用来获取 Servlet API 中的 OutputStream/Writer。

- HttpMethod，枚举类型，对应于 HTTP Method，如 POST、GET。
- @MatrixVariable，矩阵变量。
- @RequestParam，对应于 HTTP 请求的参数，自动转化为参数对应的类型。
- @RequestHeader，对应于 HTTP 请求头参数，自动转化为对应的类型。
- @RequestBody，自动将请求内容转为指定的对象，默认使用 HttpMessageConverters 来转化。
- @RequestPart，用于文件上传，对应于 HTTP 协议的 multipart/form-data。
- @SessionAttribute，该方法标注的变量来自于 Session 的属性。
- @RequestAttribute，该标注的变量来自于 request 的属性。
- @InitBinder，用在方法上，说明这个方法会注册多个转化器，用来个性化地将 HTTP 请求参数转化成对应的 Java 对象，如转化为日期类型、浮点类型、JavaBean 等，当然，也可以实现 WebBindingInitializer 接口来用于 Spring Boot 应用所需要的 dataBinder。
- BindingResult 和 Errors，用来处理绑定过程中的错误。

这里部分含义比较明确，限于篇幅不做详细讲解，下面会对常用的 PathVariable、Model、ModelAndView、JavaBean、文件上传、ModelAttribute 进行讲解。

## 3.4.1 PathVariable

注解 PathVariable 用于从请求 URL 中获取参数并映射到方法参数中，如以下代码所示。

```
@Controller
@RequestMapping("/user/{id}")
public class Sample35Controller {
  @Autowired UserService userService;

  @GetMapping(path = "/{type}/get.json")
  @ResponseBody
  public User getUser(@PathVariable Long id,@PathVariable Integer type) {
    return userService.getUserById(id);
  }
}
```

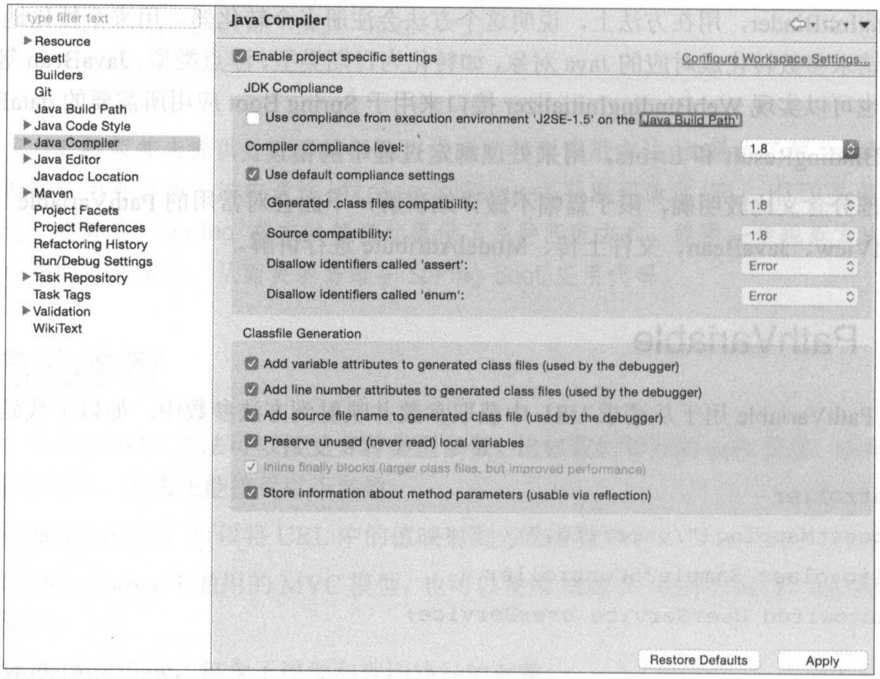
符号{}中的变量名与方法名字一一对应，如果不想对应，如 Path 中的名字是 id，方法签名

是 userId，则可以使用@PathVariable("id") Long userId 来对应。

通常情况下，Java 编译代码的时候，会将参数名称也编译到 class 字节码里，因此 Spring 会根据名字匹配自动映射。如果上述例子中出现如下错误：

```
There was an unexpected error (type=Internal Server Error, status=500).
Name for argument type [java.lang.Long] not available, and parameter name
information not found in class file either
```

则表明你的编译环境并未将调试信息加入 class 中，建议将编译器改成默认设置。以笔者使用的 Eclipse 为例，选择工程，右键选择属性，找到 Java Compiler，在 Classfile Generation 选项中勾选除 Inline finally Blocks 外的所有选项，如下图所示。




Spring 也支持 URL 中的矩阵变量，所谓矩阵变量，就是出现在路径片段中，通过符号“;”分割的多个变量，比如/user/id=123;status=1/update.json。

### 3.4.2 Model&ModelAndView

任何 MVC 框架都有一个类似 Map 结构的 Model，可以向 Model 添加视图需要的变量，Spring MVC 中的 Model 就是完成此功能的。Model 对象主要有如下方法：

- Model addAttribute(String attributeName, Object attributeValue)，向模型添加一个变量，attributeName 指明了变量的名称，可以在随后的视图里引用，attributeValue 代表了变量。
- Model addAttribute(Object attributeValue)，向模型添加一个变量，变量的名字就是其类名字首字母小写后转为的 Java 变量。
- Model addAllAttributes(Map attributes)，添加多个变量，如果变量已经存在，则覆盖。
- Model mergeAttributes(Map attributes)，添加多个变量，如果变量已经存在于模型中，则忽略。
- Model addAllAttributes(Collection<?> attributeValues)，添加多个变量，变量来自于集合的元素，变量命名规范同 addAttribute(Object attributeValue)。
- boolean containsAttribute(String attributeName)，判断是否存在变量。

Model 用于参数的时候，Spring MVC 框架在调用方法前自动创建 Model，如以下实例所示。

```
@GetMapping(path = "/{userId}/get.html")
public String getUser(@PathVariable Long userId,Model model) {
  User userInfo =  userService.getUserById(userId);
  //model.addAttribute(userInfo); 与下面代码效果相同
  model.addAttribute("user", userInfo);
  return "/userInfo.html";
}
```

这样可以在视图页面中通过“user”来访问其值，比如在 userInfo.html 中（该文件位于 resources/templates 目录下，视图技术将在第 4 章详细说明）：

```
<html>
<body>
${user.id}
<p/>
${user.name}
</body>
</html>
```

> 在这个例子中，也可以使用 model.addAttribute(userInfo)，相当于 model.addAttribute("user",userInfo)，但不建议这么做，因为 key 值是根据类名来确定的，类名重构后，会导致视图渲染失败。

ModelAndView 对象类似 Model，但额外提供了一个视图名称，因此上述代码也可以改成如下代码：

```
@GetMapping(path = "/{userId}/get2.html")
public ModelAndView getUser2(@PathVariable Long userId,ModelAndView view) {
  User userInfo =  userService.getUserById(userId);
  view.addObject("user", userInfo);
  view.setViewName("/userInfo.html");
  return view;
}
```

ModelAndView 对象既可以通过方法声明，也可以在方法中构造，上面的例子也可以写成：

```
@GetMapping(path = "/{userId}/get2.html")
public ModelAndView getUser2(@PathVariable Long userId) {
  ModelAndView view = new ModelAndView();
  ......
  return view;
}
```

### 3.4.3 JavaBean 接受 HTTP 参数

HTTP 提交的参数可以映射到方法参数上，按照名称来映射，比如一个请求/javabean/update2.json?name=abc&id=1，将会调用以下方法：

```
@GetMapping(path = "/update2.json")
@ResponseBody
public String getUser2(Integer id, String name) {

  return "success";
}
```

可以通过注解@RequestParam 来进一步限定 HTTP 参数到 Controller 方法的映射关系，RequestParam 支持如下属性：

- value，指明 HTTP 参数的名称。
- required，boolean 类型，声明此参数是否必须有，如果 HTTP 参数里没有，则会抛出 400 错误。

- defaultValue，字符类型，如果 HTTP 参数没有提供，可以指定一个默认字符串，Spring 类型转化为目标类型，如上一个例子，我们可以提供默认参数。

```
public String getUser2(@RequestParam(name="id",required=true) Integer id,
String name)
```

可以将 HTTP 参数转为 JavaBean 对象，HTTP 参数的名字对应到 POJO 的属性名：

```
@GetMapping(path = "/update.json")
@ResponseBody
public String updateUser(User user) {
  return "success";
}
```

通常，HTTP 提交了多个参数，Spring 支持按照前缀自动映射到不同的对象上。比如用户提交了订单信息，订单信息包含了多个订单明细。

需要创建一个 Form 对象来接收 HTTP 提交的数据：

```
public class OrderPostForm {
    private Order order;
    private List<OrderDetail> details;
    // 忽略 getter 和 setter
}
```

如上所示，以“order”为前缀的 HTTP 参数将映射到 OrderPostForm 类的 order 属性上，以 details 属性为前缀的 HTTP 参数将映射到 OrderPostForm 的 details 属性上。如果提交的表单如下：

```
<form action="/javabean/saveOrder.json" method="post">
订单名称:<input name="order.name"/>
<p>
订单明细 1
<input name="details[0].name" >
订单明细 2
<input name="details[1].name" >
<p>
<input type="submit" value="Submit" />
</form>
```

则下面的 Controller 的方法可以处理这个请求：

```
@PostMapping(path = "/saveorder.json")
@ResponseBody
public String saveOrder( OrderPostForm form) {
  return "success";
}
```

简单来说，Spring 有如下表所示的 HTTP 参数到 JavaBean 的映射规则。

| 示　　例 | 解　　释 |
|---|---|
| name | 对象的 name 属性 |
| order.name | 对象的 order 属性的 name 属性 |
| details[0].name | 对象的 details 属性，要求 details 是个数组或者 List（不能是 Set，因为 Set 不具备根据索引取值的功能），details[0]表示 details 属性的第一个元素 |

## 3.4.4 @RequsetBody 接受 JSON

Controller 方法带有@RequsetBody 注解的参数，意味着请求的 HTTP 消息体的内容是一个 JSON，需要转化为注解指定的参数类型。Spring Boot 默认使用 Jackson 来处理反序列化工作。

相对于 3.4.3 节中使用 JavaBean 来接受 HTTP 提交的参数，如果客户端发起了一个 JSON 请求，则有如下定义来接受 JSON：

```
@PostMapping(path = "/savejsonorder.json")
@ResponseBody
public String saveOrderByJson(@RequestBody User user) {
  return user.getName();
}
```

这段代码能处理客户端发起的 JSON 请求，这里使用 curl 命令发起一个请求：

```
>curl -XPOST 'http://127.0.0.1:8080/javabean/savejsonorder.json' -H
'Content-Type: application/json' -d'
  {
    "name":"hello",
    "id":1
```

```
}
'
```

> 事实上，本章的大部分 HTTP 请求都不是采用浏览器或者编写一个前端代码来发起的，而是用 curl 命令来发起的。如上述 curl 命令，将发起一个 POST 请求，且用-H 参数设置 HTTP 头，用-d 参数设置请求体的内容。curl 命令在 Linux、Mac 系统里都是内置的，Windows 系统则需要自己下载安装，如果你有兴趣使用 curl 来测试 Controller，则可以先阅读 3.10 节 curl 部分的内容。

## 3.4.5 MultipartFile

通过 MultipartFile 来处理文件上传：

```
@PostMapping("/form")
@ResponseBody
public String handleFormUpload( String name,
                                MultipartFile file) throws IOException {
  if (!file.isEmpty()) {
    String fileName = file.getOriginalFilename();
    InputStream ins = file.getInputStream();

    // 处理上传内容
    return "success";
  }
  return "failure";
}
```

MultipartFile 提供了以下方法来获取上传的文件信息：

- getOriginalFilename，获取上传的文件名字；
- getBytes，获取上传文件内容，转为字节数组；
- getInputStream，获取一个 InputStream；
- isEmpty，文件上传内容为空，或者就没有文件上传；
- getSize，文件上传的大小；
- transferTo(File dest)，保存上传文件到目标文件系统。

如果是同时上传多个文件，则使用 MultipartFile 数组类来接受多个文件上传：

```
@PostMapping("/form")
@ResponseBody
public String handleFormUpload( String name,
                                MultipartFile[] files) throws IOException {
}
```

这要求你的 HTTP 请求中包含多个名字为“files”的文件：

```
<form action="filesUpload.html" method="post"
      enctype="multipart/form-data">
  <p>
   选择文件:<input type="file" name="files">
  <p>
   选择文件:<input type="file" name="files">
  <p>
   选择文件:<input type="file" name="files">
  <p>
   <input type="submit" value="提交">
</form>
```

可以通过配置文件 application.properties 对 Spring Boot 上传的文件进行限定，默认为如下配置：

```
spring.servlet.multipart.enabled=true
spring.servlet.multipart.file-size-threshold=0
spring.servlet.multipart.location=
spring.servlet.multipart.max-file-size=1MB
spring.servlet.multipart.max-request-size=10MB
spring.servlet.multipart.resolve-lazily=false
```

参数 enabled 默认为 true，即允许附件上传，file-size-threshold 限定了当上传的文件超过一定长度时，就先写到临时文件里。这有助于上传文件不占用过多的内存，单位是 MB 或者 KB，默认是 0，即不限定阈值。location 指的是临时文件的存放目录，如果不设定，则是 Web 服务器提供的一个临时目录。

max-file-size 属性指定了单个文件的最大长度，默认是 1MB，max-request-size 属性说明单

次 HTTP 请求上传的最大长度，默认是 10MB。

resolve-lazily 表示当文件和参数被访问的时候再解析成文件。

> 如果上传较大文件失败，则需要检查是不是因为 Spring Boot 对文件的限定过小造成的。另一方面，有些 Spring Boot 应用设置了代理服务器，比如设置了 Apache，也需要检查代理服务器是否支持大文件上传，是否对超时做了设定。

### 3.4.6 @ModelAttribute

注解 ModelAttribute 通常作用在 Controller 的某个方法上，此方法会首先被调用，并将方法结果作为 Model 的属性，然后再调用对应的 Controller 处理方法。

```
@ModelAttribute
public void findUserById(@PathVariable Long id,Model  model) {
  model.addAttribute("user", userService.getUserById(id));
}

@GetMapping(path = "/{id}/get.json")
@ResponseBody
public String getUser(Model model) {
  System.out.println(model.containsAttribute("user"));
  return "success";
}
```

对于 HTTP 的请求，modelattribute/1/get.json，会先调用 findUserById 方法取得 user，并添加到模型里。使用 ModelAttribute 通常可以用来向一个 Controller 中需要的公共模型添加数据。

如果 findUserById 仅仅添加一个对象到 Model 中，则可以改写成如下形式：

```
@ModelAttribute
public User findUserById(@PathVariable Long id) {
  return userService.getUserById(id);
}
```

这样，返回的对象自动添加到 Model 中，相当于调用 model.addAttribute(user)。

### 3.4.7 @InitBinder

前面我们学习了 Spring 如何将 HTTP 参数绑定到 JavaBean 对象中，Spring 框架通过 WebDataBinder 类实现这种绑定，可以在 Controller 中用注解@InitBinder 声明一个方法，来自己扩展绑定的特性，比如：

```
@Controller
public class MyFormController {

  @InitBinder
  protected void initBinder(WebDataBinder binder) {
    binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
  }

  @ResponseBody
  @RequestMapping("/date")
  public void printDate(Date d) {
    System.out.println(d);
    return ;
  }

}
```

当需要绑定到一个 Date 类型的时候，如上述代码所示，则采用“yyyy-MM-dd”格式，比如用户访问 databind/date?d=2011-1-1。

也可以定义一个全局的转化函数，会在 3.6.3 节详细讲解。

## 3.5 验证框架

Spring Boot 支持 JSR-303、Bean 验证框架，默认实现用的是 Hibernate validator。在 Spring MVC 中，只需要使用@Valid 注解标注在方法参数上，Spring Boot 即可对参数对象进行校验，校验结果放在 BindingResult 对象中。

### 3.5.1 JSR-303

JSR-303 是 Java 标准的验证框架，已有的实现有 Hibernate validator。JSR-303 定义了一系

列注解用来验证 Bean 的属性，常用的有如下几种。

- 空检查
  - @Null，验证对象是否为空；
  - @NotNull，验证对象不为空；
  - @NotBlank，验证字符串不为空或者不是空字符串，比如""和" "都会验证失败；
  - @NotEmpty，验证对象不为 null，或者集合不为空。
- 长度检查
  - @Size(min=, max=)，验证对象长度，可支持字符串、集合；
  - @Length，字符串大小。
- 数值检测
  - @Min，验证数字是否大于等于指定的值；
  - @Max，验证数字是否小于等于指定的值；
  - @Digits，验证数字是否符合指定格式，如@Digits(integer=9,fraction=2)；
  - @Range，验证数字是否在指定的范围内，如@Range(min=1, max=1000)。
- 其他
  - @Email，验证是否为邮件格式，为 null 则不做校验；
  - @Pattern，验证 String 对象是否符合正则表达式的规则。

以下是一个包含了验证注解的 JavaBean：

```
public class WorkInfoForm {
    @NotNull
    Long id;
    @Size(min=3,max=20)
    String name;
    @Email
    String email;

}
```

通常，不同的业务逻辑会有不同的验证逻辑，比如对于 WorkInfoForm 来说，当更新的时候，id 必须不为 null，但增加的时候，id 必须是 null。

JSR-303 定义了 group 概念，每个校验注解都必须支持。校验注解作用在字段上的时候，可

以指定一个或者多个 group，当 Spring Boot 校验对象的时候，也可以指定校验的上下文属于哪个 group。这样，只有 group 匹配的时候，校验注解才能生效。上面的 WorkInfoForm 定义 id 字段校验可以更改为如下内容：

```
public class WorkInfoForm {
  // 定义一个类，更新时校验组
  public interface Update{}
  // 定义另一个类，添加时校验组
  public interface Add{}

  @NotNull(groups={Update.class})
  @Null(groups={Add.class})
  Long id;
}
```

这段代码表示，当校验上下文为 Add.class 的时候，@Null 生效，id 需为空才能校验通过；当校验上下文为 Update.class 的时候，@NotNull 生效，id 不能为空。

### 3.5.2 MVC 中使用@Validated

在 Controller 中，只需要给方法参数加上@Validated 即可触发一次校验。

```
@ResponseBody
@RequestMapping("/addworkinfo.html")
public void addWorkInfo(@Validated({WorkInfoForm.Add.class}) WorkInfoForm
workInfo,BindingResult result) {
  if(result.hasErrors()){
    List<ObjectError>  list = result.getAllErrors();
    FieldError  error = (FieldError)list.get(0);
    System.out.println(error.getObjectName()+","+error.getField()+",
"+error.getDefaultMessage());
    return ;
  }
  return ;
}
```

此方法可以接受 HTTP 参数并映射到 WorkInfoForm 对象，WorkInfoForm 参考上一节说明，

此参数使用了@Validated注解，将触发Spring的校验，并将验证结果存放到BindingResult对象中。这里，Validated注解使用了校验的上下文WorkInfoForm.Add.class，因此，整个校验将按照Add.class来校验。

BindingResult包含了验证结果，提供了如下方法：

- hasErrors，判断验证是否通过；
- getAllErrors，得到所有的错误信息，通常返回的是FieldError列表。

如果Controller参数未提供BindingResult对象，则Spring MVC将抛出异常。关于如何处理Spring MVC异常，请参考3.8节通用错误处理。

> **注意**：通常情况下，Spring Boot应用的前端应该已经做了充分校验，后端校验仍然可以采用以防止用户绕过前端校验。

## 3.5.3 自定义校验

JSR-303提供的大部分校验注解已经够用，也允许定制校验注解，比如在WorkInfoForm类中，我们新增一个加班时间：

```
@WorkOverTime(max=2)
int workTime;
```

属性workTime使用了注解@WorkOverTime，当属性值超过max值的时候，将会验证失败。WorkOverTime跟其他注解差不多，但提供了@Constraint来说明用什么类作为验证注解实现类，代码如下：

```
@Constraint(validatedBy = { WorkOverTimeValidator.class })
@Documented
@Target({ ElementType.ANNOTATION_TYPE, ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface WorkOverTime {
  String message() default "加班时间过长，不能超过{max}小时";
  int max() default 5;
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

@Constraint注解声明用什么类来实现验证，我们将创建一个WorkOverTimeValidator来进

行验证。验证注解必须要提供如下信息：

- message，用于创建错误信息，支持表达式，如“错误，不能超过（max）小时”。
- groups，验证规则分组，比如新增和修改的验证规则不一样，分为两个组，验证注解必须提供。
- payload，定义了验证的有效负荷。

WorkOverTimeValidator 必须实现 ConstraintValidator 接口 initialize 方法及验证方法 isValid：

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class WorkOverTimeValidator implements
ConstraintValidator<WorkOverTime, Integer>{
  WorkOverTime work;
  int max ;
  public void initialize(WorkOverTime work) {
    // 获取注解的定义
    this.work = work;
    max = work.max();
  }

  public boolean isValid(Integer value, ConstraintValidatorContext context) {
    // 校验逻辑
    if(value==null){
      return true;
    }
    return value<max;
  }
}
```

# 3.6 WebMvcConfigurer

WebMvcConfigurer 是用来全局定制化 Spring Boot 的 MVC 特性。开发者通过实现 WebMvcConfigurer 接口来配置应用的 MVC 全局特性。

```
@Configuration
public class MvcConfigurer implements WebMvcConfigurer {
```

```
    // 拦截器
    public void addInterceptors(InterceptorRegistry registry) {

    }
    // 跨域访问配置
    public void addCorsMappings(CorsRegistry registry) {
    }
    // 格式化
    public void addFormatters(FormatterRegistry registry) {


    }
    // URI 到视图的映射
    public void addViewControllers(ViewControllerRegistry registry) {
    }

    // 其他更多全局定制接口
}
```

## 3.6.1 拦截器

通过 addInterceptors 方法可以设置多个拦截器，比如对特定的 URI 设定拦截器以检查用户是否登录，打印处理用户请求耗费的时间等。

```
public void addInterceptors(InterceptorRegistry registry) {
  // 增加一个拦截器，检查会话，URL 以 admin 开头的都使用此拦截器
  registry.addInterceptor(new
SessionHandlerInterceptor()).addPathPatterns("/admin/**");
}

class SessionHandlerInterceptor implements HandlerInterceptor{
  public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
    throws Exception {
    User user = (User) request.getSession().getAttribute("user");
    if(user==null){
      // 如果没有登录，重定向到 login.html
```

```
        response.sendRedirect("/login.html");
         return false;
        }
        return true;
    }

    public void postHandle(
        HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView)
        throws Exception {
        // Controller 方法处理完毕后，调用此方法
    }
    @Override
    public void afterCompletion(
        HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex)
        throws Exception{
        // 页面渲染完毕后调用此方法，通常用来清除某些资源，类似 Java 语法的 finally
    }
}
```

上面这段代码中，我们在 MvcConfigurer 中实现了 addInterceptors 方法，并对访问地址 /admin/**添加了一个拦截器。关于 URL 匹配可以参考 3.3.2 节 URL 路径匹配。

拦截器有以下三个方法需要覆盖实现：

- preHandle，在调用 Controller 方法前会调用此方法。
- postHandle，在调用 Controller 方法结束后、页面渲染之前调用此方法，比如可以在这里将渲染的视图名称更改为其他视图名称。
- afterCompletion，页面渲染完毕后调用此方法。

## 3.6.2 跨域访问

出于安全的考虑，浏览器会禁止 AJAX 访问不同域的地址。W3C 的 CORS 规范（Cross-origin resource sharing）允许实现跨域访问，并被现在大多数浏览器支持，包括：

- Chrome 3+；
- Firefox 3.5+；

- Opera 12+;
- Safari 4+;
- Internet Explorer 8+。

Spring Boot 提供了对 CORS 的支持，可以实现 addCorsMappings 接口来添加特定的配置：

```
@Override
public void addCorsMappings(CorsRegistry registry) {
  registry.addMapping("/**");
}
```

允许所有跨域访问，或者更为精细的控制，如下所示。

```
public void addCorsMappings(CorsRegistry registry) {
  registry.addMapping("/api/**")
   .allowedOrigins("http://domain2.com")
   .allowedMethods("POST", "GET");

}
```

仅仅允许来自 domain2.com 的跨域访问，并且限定访问路径为/api、方法是 POST 或者 GET。

> 跨域原理简单点理解就是发起跨域请求的时候，浏览器会对请求域返回的响应信息检查 HTTP 头，如果 Access-Control-Allow-Origin 包含了自身域，则表示允许访问。否则报错，这就是 allowedOrigins 的作用。

### 3.6.3 格式化

将 HTTP 请求映射到 Controller 方法的参数上后，Spring 会自动进行类型转化。对于日期类型的参数，Spring 默认并没有配置如何将字符串转为日期类型。为了支持可按照指定格式转为日期类型，需要添加一个 DateFormatter 类：

```
public void addFormatters(FormatterRegistry registry) {
  registry.addFormatter(new DateFormatter("yyyy-MM-dd HH:mm:ss"));
}
```

DateFormatter 类实现将字符串转为日期类型 java.util.Data。

### 3.6.4 注册 Controller

应用有时候没有必要为一个 URL 指定一个 Controller 方法，可以直接将 URI 请求转到对模板的渲染上。比如应用中的如下代码：

```
@RequestMapping("/")
public  String index(){
  return "/index.btl";
}
```

可以直接通过 ViewControllerRegistry 注册一个：

```
public void addViewControllers(ViewControllerRegistry registry) {
  registry.addViewController("/index.html").setViewName("/index.btl");
  registry.addRedirectViewController("/**/*.do", "/index.html");
}
```

对于 index.html 的请求，设置返回的视图为 index.btl。

所有以.do 结尾的请求重定向到/index.html 请求。

## 3.7 视图技术

Spring Boot 支持多种视图技术，内置支持如下：

- FreeMarker；
- Groovy；
- Thymeleaf；
- Mustache。

本章将简要介绍如何在 Spring 中使用模板技术，并会在第 4 章重点介绍笔者开发的 Beetl 模板引擎。

### 3.7.1 使用 Freemarker

安装 Freemaker，需要在 pom 中添加如下依赖：

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
```

Spring Boot 模板都假定在 resources/templates 下，因此模板文件需要创建在其目录下，我们创建一个简单的模板 test.ftl：

```
<html>
<body>
${user.id}
<p/>
${user.name}
</body>
</html>
```

默认情况下，Spring Boot 的 Freemarker 模板引擎为视图添加 ftl 后缀来寻找匹配的模板，在 Controller 中，我们使用 ModelAndView：

```
@Controller
@RequestMapping("/freemarker")
public class FreemakerController {
  @AuTwoired UserService userService;
  @GetMapping("/showuser.html")
  public ModelAndView showUserInfo(Long id){
    ModelAndView view = new ModelAndView();
    User user = userService.getUserById(id);
    view.addObject("user",user);
    view.setViewName("/userInfo ");
    return view;
  }
}
```

访问/freemarker/showuser.html?id=1，Spring Boot 将调用 showUserInfo 方法，该方法创建一个 ModelAndView，并设置视图名称为“/userInfo”，Freemarker 会自动添加后缀 ftl 来寻找匹配的 userInfo.ftl 模板。

> **注意：** Spring Boot 支持同时使用多种模板引擎，通常情况下，Spring Boot 会依次询问每种模板引擎是否能处理指定的视图名。对于 Freemarker，默认情况下会自动加上 ftl 后缀，然后再判断是否存在此模板文件。

### 3.7.2 使用 Beetl

Beetl 是笔者开发的一个模板引擎，它的语法和使用习惯参考了 JS，因此学习起来较快，同时在功能和性能上也做了很大的提升。集成 Beetl 需要在 Maven 中添加如下依赖：

```
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl-framework-starter</artifactId>
  <version>1.1.6.RELEASE</version>
</dependency>
```

beetl-framework-starter 自动识别以 btl 结尾的视图名称并交给 Beetl 模板引擎渲染。

这时我们可以创建一个 Controller 来进行测试：

```
@Controller
@RequestMapping("/beetl")
public class BeetlController {
  @Autowired UserService userService;
  @GetMapping("/showuser.html")
  public ModelAndView showUserInfo(Long id){
    ModelAndView view = new ModelAndView();
    User user = userService.getUserById(id);
    view.addObject("user",user);
    view.setViewName("/userInfo.btl");
    return view;
  }
}
```

如果访问/beetl/showuser.html?id=1，则 Spring Boot 会调用 showUserInfo 方法，该方法创建了一个 ModelAndView，用来设置视图名称为"/userInfo.btl"，Spring Boot 看到视图名称后缀是 btl，就使用 Beetl 模板引擎进行渲染，即渲染 templates/userInfo.btl 模板文件。

> 关于更多 Beetl 作为后端模板引擎的内容，将放到本书第 4 章视图技术的第一部分详细讲述。

### 3.7.3 使用 Jackson

在MVC框架中，Spring Boot内置了Jackson来完成JSON的序列化和反序列化。在Controller中，方法注解为@ResponseBody，自动将方法返回的对象序列化成JSON。

```
@Controller
@RequestMapping("/json")
public class BeetlController {
    @Autowired UserService userService;
    @GetMapping("/user/{id}.json")
    public @ResponseBody User showUserInfo( @PathVariable Long id){
        User user = userService.getUserById(id);
        return user;
    }
}
```

如果想自己全局自定义一个ObjectMapper来代替默认的，则可以使用Java Config，联合使用@Bean，代码如下：

```
@Configuration
public class JacksonConf {

  @Bean
  public ObjectMapper getObjectMapper() {
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    return objectMapper;
  }
}
```

上述Java Config会使Spring Boot使用自定义的Jackson来序列化而非默认配置的。以下是一个用来获取当前时间的请求：

```
@GetMapping("/now.json")
public @ResponseBody Map now(){
  Map map = new HashMap();
  map.put("time", new Date());
```

```
  return map;
}
```

当用户访问 now.json 的时候，会得到如下输出：

```
{"time":"2017-04-12 22:52:05"}
```

> 关于 Jackson 的配置，还有序列化和反序列化，我们将在第 4 章视图技术的第二部分详细讲述。

### 3.7.4 Redirect 和 Forward

通常而言，Controller 都会返回一个视图名称，比如以 btl 结尾的视图会交给 Beetl 模板引擎渲染。有些情况下，Controller 会返回客户端一个 HTTP Redirect 重定向请求，希望客户端按照指定地址重新发起一次请求，比如客户登录成功后，重定向到后台系统首页。再比如客户端通过 POST 提交了一个名单，可以返回一个重定向请求到此订单明细的请求地址。这样做的好处是，如果用户再次刷新页面，则访问的是订单详情地址，而不会再次提交订单。

Controller 中重定向可以返回以“redirect:”为前缀的 URI：

```
@RequestMapping("/order/saveorder.html")
public String saveOrder(Order order){
  Long orderId  = service.addOrder(order)
  return "redirect:/order/detail.html?orderId="+orderId;
}
```

还可以在 ModelAndView 中设置带有“redirect:”前缀的 URI：

```
ModelAndView view = new ModelAndView("redirect:/order/detail.html?orderId=
"+orderId);
```

或者直接使用 RedirectView 类：

```
RedirectView  view = new RedirectView("/order/detail.html?orderId=
"+orderId);
```

Spring MVC 也支持 foward 前缀，用来在 Controller 执行完毕后，再执行另外一个 Controller 的方法。

```
@RequestMapping("/bbs")
public String index(){
  // forward 到 module 方法
  return "forward:/bbs/module/1-1.html;
}

@RequestMapping("/bbs/moudle/{type}-{page}")
public ModelAndView  module(@PathVariable int type,PathVariable int page){
  ....
}
```

对于所有访问/bbs 的请求，都会 foward 到 module 方法，因为 foward 的 URL 是/bbs/module/1-1.html，正好匹配 module 方法的@RequestMapping 的定义。

## 3.8 通用错误处理

在 Spring Boot 中，Controller 中抛出的异常默认交给了/error 来处理，应用程序可以将/error 映射到一个特定的 Controller 中处理来代替 Spring Boot 的默认实现，应用可以继承 AbstractErrorController 来统一处理系统的各种异常。

```
@Controller
public class ErrorController extends AbstractErrorController {

  Log log = LogFactory.getLog(ErrorController.class);
  @Autowired
  ObjectMapper objectMapper;

  public ErrorController() {
     super(new DefaultErrorAttributes());
  }

  @RequestMapping("/error")
  public ModelAndView getErrorPath( HttpServletRequest request,
HttpServletResponse response) {
     // 处理异常
  }
}
```

AbstractErrorController 提供了多个方法可以从 request 中获取错误信息，包含以下信息：

- timestamp，错误发生的时间；
- status，对应于 HTTP Status，如 404；
- error，错误消息，如 Bad Request、Not Found；
- message，详细错误信息；
- exception，如果应用抛出有异常，exception 是字符串，代表异常的类名，如 org.springframework.web.method.annotation.MethodArgumentTypeMismatchException；
- path，请求的 URI；
- errors，@Validated 校验错误的时候，校验结果信息放到这里，参考 3.5.2 节 MVC 中使用@Validated。

考虑到异常信息直接显示给应用系统客户并不合适，尤其是 RuntimeException。同时，还要区分页面渲染和 JSON 请求这两种不同的情况，前者应该返回一个错误页面，后者应该返回一个 JSON 结果。因此，为应用提供的统一错误处理代码应该如下：

```
  @RequestMapping(ERROR_PATH)
  public ModelAndView getErrorPath(HttpServletRequest request,
HttpServletResponse response) {
    Map<String, Object> model =
Collections.unmodifiableMap(getErrorAttributes(
      request, false));
    // 获取异常，有可能为空
    Throwable cause =getCause(request);
    int status =  (Integer)model.get("status");
    // 错误信息
    String message = (String)model.get("message");
    // 友好提示
    String errorMessage = getErrorMessage(cause);
    // 后台打印日志信息方便查错
    log.info(status+","+message, cause);
    response.setStatus(status);
    if(!isJsonRequest(request)){
      // error.btl 模板显示错误的详细信息
      ModelAndView view = new ModelAndView("/error.btl");
      view.addAllObjects(model);
```

```
      view.addObject("errorMessage", errorMessage);
      view.addObject("status", satus);
      view.addObject("cause", cause);
      return view;

    }else{
      Map error = new HashMap();
      error.put("success", false);
      error.put("errorMessage", errorMessage);
      error.put("message", message);
      writeJson(response,error);
      return null;
    }
  }
```

getErrorAttributes 方法是 AbstractErrorController 提供的用于获取错误信息的方法，返回一个 Map，包含的数据如下所述。

getCause 方法用于获取应用系统的异常，定义如下：

```
  protected Throwable getCause(HttpServletRequest request) {
    Throwable error =
(Throwable)request.getAttribute("javax.servlet.error.exception");
    if (error != null) {
      // MVC 有可能会封装异常成 ServletException，需要调用 getCause 获取真正的异常
      while (error instanceof ServletException && error.getCause() != null) {
        error = ((ServletException) error).getCause();
      }
    }
    return error;
  }
```

getErrorMessage 方法返回一个友好的异常信息，而不是 Spring Boot 提供的 message 包含的信息：

```
  protected String getErrorMessage(Throwable ex) {
    return "服务器错误,请联系管理员";
  }
```

通常这个友好信息很简单，类似上面的“服务器错误，请联系管理员”，也可以根据应用

系统自定义的异常来进一步输出详细信息，比如：

```
protected String getErrorMessage(Throwable ex) {
  if(ex instanceof YourApplicationException){
    // 如果 YourApplicationException 的信息可以显示给用户
    return  ((YourApplicationException)ex).getMessage();
  }
  return "服务器错误,请联系管理员";
}
```

isJsonRequest 方法用来区分客户端发起的是页面渲染请求还是 JSON 请求，定义如下：

```
protected boolean isJsonRequest(HttpServletRequest request){
  String requestUri =
(String)request.getAttribute("javax.servlet.error.request_uri");
  if(requestUri!=null&&requestUri.endsWith(".json")){
    return true;
  }else{
    // 也可以通过获取 HTTP 头，根据 Accept 字段是否包含 JSON 来进一步判断，比如
    // request.getHeader("Accept").contains("application/json")
    return false;
  }
}
```

## 3.9 @Service 和@Transactional

到目前为止，Spring Boot 的 Controller 介绍完毕，在 Spring Boot 中，Controller 调用业务逻辑处理交给了被@Service 注解的类，这也是个普通的 JavaBean，Controller 中可以自动注入这种 Bean，并调用其方法完成主要的业务逻辑。正如 Controller 注解经常和@RequestMapping 搭配使用一样，@Service 和@Transactional 配合使用。

### 3.9.1 声明一个 Service 类

在 Spring Boot 应用中，业务逻辑集中在 Service 中处理，并自动注入到 Controller 中处理。实现一个 Service 类，需要定义一个业务的接口，比如根据 id 查询用户及更新用户状态：

```
package com.bee.sample.ch3.service;
public interface UserService {

  public User getUserById(Long id);
  public void updateUser(Long id,Integer type);
}
```

然后实现此业务接口，不要忘记增加@Service来引起Spring Boot的“注意”，同时搭配上@Transactional，Spring Boot会对这样的Bean进行事务增强。

```
package com.bee.sample.ch3.service.impl;

@Service
@Transactional
public class UserServiceImpl implements UserService {

  public User getUserById(Long id) {

      return user;
  }

  public void updateUser(Long id, Integer type) {
  }
}
```

> 尽管Spring Boot不要求Service必须实现某个接口，但笔者建议还是为Service定义接口，这样可以允许为这个Service提供不同的实现，比如模拟业务实现，单元测试中通过Mock来模拟一个Service实现。

## 3.9.2 事务管理

Spring简单地实现了事务管理，通过在Service类中使用@Transanal来让Service参与事务管理。为了使用事务，需要在pom中添加以下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

@Transactional 可以作用在类上，这样，所有的接口方法都会参与事务管理。也可以放到方法上，上述的 update 方法可以使用@Transactional 来声明调用该方法会处于事务上下文中。

```
@Transactional
public void updateUser(Long id, Integer type) {

}
```

当 Controller 调用 Service 方法的时候，会开启一个事务上下文，随后的调用都将处于这个事务上下文中。如果调用这个 Service 方法抛出 RuntimeException，事务会自动回滚。否则，事务将提交。

> 事务上下文：对于 Service 调用，如果处于同一个事务上下文，那么对数据库的操作会在一个事务中。事务上下文的开启是从 Controller 中调用 Service 方法的时候自动开启的，并在调用此方法后自动结束从而提交事务，或者根据抛出的 RuntimeException 来自动回滚，调用过程中调用的其他 Service 方法都会处于这个事务上下文中。
>
> 可以在事务上下文中再次开启一个新的事务上下文，这时候通过注解@Transactional (propagation = Propagation.REQUIRES_NEW)完成，比如在 AuditService 中，审计服务无论业务调用是否成功，都必须把审计信息存储到数据库中，因此 AuditService 可以配置为开启新的事务上下文。

## 3.10 curl 命令

curl 是利用 URL 语法在命令行方式下工作的开源文件传输工具。它被广泛应用在 UNIX、多种 Linux 发行版中，并且有 DOS 和 Win32、Win64 下的移植版本。如果你的开发环境是 Mac 或者 Linux，会自带 curl；如果是 Windows 系统，需要从 https://curl.haxx.se/下载 window 版本；如果你安装了 git shell，也自带了 curl。

curl 命令可以在我们开发 Web 应用的时候，模拟前端发起的 HTTP 请求，本节简要介绍 curl 命令，这一章和以后各章的 Controller 代码测试可以使用 curl 发起测试，而且能很好地模拟 POST、PUT 等其他协议的测试，文件上传测试等用浏览器无法直接测试的也可以使用 curl 命令。

curl 最简单的命令是 curl URL，以下输入将返回请求地址的内容：

```
> curl baidu.com
<html>
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
</html>
```

通过-i 参数返回，还返回 HTTP 头：

```
curl -i baidu.com
HTTP/1.1 200 OK
Date: Sun, 09 Jul 2017 14:11:59 GMT
Server: Apache
Last-Modified: Tue, 12 Jan 2010 13:48:00 GMT
ETag: "51-47cf7e6ee8400"
Connection: Keep-Alive
Content-Type: text/html
......
<html>
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
</html>
```

通过-H 设置请求的 HTTP 头，比如请求体是 JSON 格式：

```
>curl URL -H 'Content-Type: application/json'
```

URL 通常用双引号防止转义，比如&符号在命令行中表示后台运行，因此这里必须用引号：

```
> curl "baidu.com?q=txt&c=1"
```

通过-d 参数发起 POST 请求，-d 后面是 POST 的内容：

```
>curl URL -d "param1=value1&param2=value2"
```

如果 POST 内容需要转义，比如中文字符、空格等，可以使用--data-urlencode：

```
>curl URL --data-urlencode "param1=value1&param2=中文"
```

-G 参数发起一个 GET 请求，可以联合-data-urlencode 来转义 URL 参数里的中文特殊符号，data-urlencode 默认是 POST 请求，如果没有-G 参数，则会发起一个 POST 请求：

```
>curl  -G "baidu.com"  --data-urlencode "param1=value1&param2=中文"
```

以上命令对应了一个请求 baidu.com?param1=value1&param2=中文。

发起一个 JSON 请求，通过-X 指定 PUT 协议，JSON 内容可以用引号括起来：

```
>curl -XPUT 'localhost:9200/product/book/1?pretty' -H 'Content-Type:
application/json' -d'
  {
    "name" : "北京 100 种小吃",
    "type":"food",
    "postDate" : "2009-11-15",
    "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚等"
  }
  '
```

使用-F 上传文件：

```
>curl url  -F "file=@xxx.doc" -F "name=xiandafu"
```

这样，可以通过 name 字段获取提交的名字，通过 file 字段获得 profile.jpg 文件。

> Chrome 插件 Postman 能可视化地完成像 curl 这样的工作，如果不喜欢 curl 命令行方式，推荐使用 Postman。

# 第 4 章 视图技术

本章介绍 MVC 中的后端视图技术，一种是后端模板引擎 Beetl，用于渲染模板；另外一种是 JSON 序列化技术 Jackson。

本章选择介绍 Beetl，主要考虑到一方面笔者作为 Beetl 的开发者，对其非常熟悉；另外一方面，Beetl 普遍应用在国内顶尖互联网公司、大中型企业，具有易使用和性能良好等特点，有非常好的国内口碑。

对于 JSON 的序列化和反序列化技术，也有很多工具可以采用，如国内的 Fastjson，国外的 Jackson、Gson。Jackson 是 Spring Boot 内置的，也是 Spring Boot 相关很多开源产品内置的序列化工具，与 Beetl 一样，同样易使用、性能良好，堪称 JSON 首选工具。因此本章将在 4.13 节介绍 Jackson。

本章最后一节将使用 Beetl 来实现 MVC 分离开发，在前后端开发人员协商好接口后，前端人员用 Beetl 脚本来模拟后端提供的 JSON 或者 ModelAndView，从而独立完成视图的开发。

## 4.1 Beetl 模板引擎

Beetl 是笔者在 2010 年开发并维护至今的一个模板引擎，具有如下特点：

- 功能完备。作为主流模板引擎，Beetl 具有相当多的功能和其他模板引擎不具备的功能，适用于各种应用场景，比如对响应速度有很高要求的大型网站，功能繁多的 CMS 管理系统，以及代码生成器等。Beetl 本身还具有很多独特的功能来完成模板的编写和维护。

- 语法和使用习俗简单：类似 JavaScript 语法和习俗，但又专门为模板渲染定制，也支持 *HTML* 标签，使得开发 *CMS* 系统变得比较容易。
- 超高的性能。Beetl 远超过主流 Java 模板引擎性能，引擎性能 5～6 倍于 Freemarker，2 倍于 JSP。
- 易于整合。Beetl 能很容易地与各种 Web 框架整合，如 Spring MVC、ACT，国内的 Nutz 和 JFinal，还有 Struts、Jodd、Servlet 等。
- 支持模板单独开发和测试。在 MVC 架构中，即使没有 M 和 C 部分，也能开发和测试模板。
- 扩展和个性化。Beetl 支持自定义方法、格式化函数、虚拟属性、标签和 HTML 标签。同时也支持自定义占位符和控制语句起始符号。

### 4.1.1 安装 Beetl

在 pom 文件中添加以下依赖：

```
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl-framework-starter</artifactId>
  <version>1.1.15.RELEASE</version>
</dependency>
```

在 Spring Boot 中，beetl-framework-starter 将自动配置以 btl 结尾的所有视图，将自动使用 Beetl 渲染相应的 resources/templates 目录下的视图文件。

### 4.1.2 设置定界符号和占位符

Beetl 支持自定义定界符号和占位符号，默认使用<% %>作为定界符号，使用${}作为占位符号，也可以配置自己喜爱的占位符号，常用的有：

- @，和回车作为定界符号；
- <? ?>，类似 PHP 的符号；
- <!—# —>，使用 HTML 注释符号作为定界符号，加了一个#符号以区别正常的 HTML 注释。

可以通过配置文件来设置定界符号，需要在 resources 目录下创建一个 beetl.properties 文件，

设置内容如下：

```
DELIMITER_PLACEHOLDER_START=${
DELIMITER_PLACEHOLDER_END=}
DELIMITER_STATEMENT_START=@
DELIMITER_STATEMENT_END=
```

本章后面都将采用“@”和“回车换行”作为定界符号，占位符使用传统的“${”和“}”。

## 4.1.3 配置 Beetl

Beetl 为了提高渲染性能，会在渲染模板后，缓存模板的语法解析结果，Beetl 每次渲染前都会检测模板文件是否更新，如果已经更新，则重新解析模板。

由于检测模板是否更新会有一次 I/O 操作，因此线上系统可以取消检测，需要在 application.properties 中添加以下配置：

```
beetl-beetlsql.dev = false
```

> Beetl 默认配置时自动检测模板是否变化，但有的 IDE 并不会将 resource/templates 目录下的文件变化同步到 Maven 工程的 target 目录下，所以即使文件发生变化，Beetl 也检测不到。如果出现这种情况，一个通用的办法是将 resource 目录设定为 src 目录，这样 resource 目录下的任何文件改变都会同步到 target 目录下。

在 Spring Boot 应用中，所有以 btl 结尾的模板都会交给 Beetl 模板引擎渲染，如果你的模板更喜欢以 html 结尾，需要在 application.properies 中添加以下配置项：

```
beetl.suffix = html
```

## 4.1.4 groupTemplate

groupTemplate 类是 Beetl 的核心类，用来渲染模板、提供扩展函数等，通常情况下，不需要了解 groupTemplate。但如果想对 groupTemplate 进行定制，则可以通过自动注入的方式实现：

```
@Configuration
public class BeetlExtConfig {
```

```
    @Autowired GroupTemplate groupTemplate;
    @Autowired SimpleFunction simple
    @PostConstruct
    public void config(){
        groupTemplate.registerFunction("hi",simple);
    }
}
```

@PostConstruct 作用在 config 方法上，因此 Spring 会在启动阶段调用此方法，可以完成 groupTemplate 的扩展。

# 4.2 使用变量

## 4.2.1 全局变量

全局变量即通过 Model 或者 ModelAndView 传入的变量，可以在模板或者子模板中使用。在 Controller 中通过 Model、ModelAndView，或者直接使用 request 设置的变量，都可以在模板中使用，比如：

```
ModelAndView view = new ModelAndView("/index.btl");
view.addObject("user",user);
return view;
```

然后可以在 Beetl 模板中使用 user 变量。

```
<span>${user.name}</span>
```

对于 Spring Boot 应用来说，Beetl 已经提供了以下默认的全局变量：

- request 中所有的 attribute 在模板中可以直接通过 name 来引用，比如在 Controller 层 request.setAttribute("user",user)，则在模板中可以直接用${user.name}。这里也包括 Model 或者 ModelAndView 中的所有变量，都会成为模板的全局变量。
- Session 提供了 Session 会话，模板通过 session["name"]或者 session.name 引用 Session 中的变量。注意，Session 并非 Servlet 中的标准 Session 对象。参考 Servlet 来获取 HTTPSession。
- request，标准的 HTTPServletRequest，可以在模板中引用 request 属性（getter），如

${request.requestURL}。

- parameter，读取用户提交的参数，如${parameter.userId}。
- ctxPath，Web 应用中的 ContextPath。在 Spring Boot 中，应用默认是“/”。通过 application.properties 的属性 server.context-path 可以配置。
- servlet，WebVariable 的实例，包含了 HTTPSession、HTTPServletRequest、HTTPServlet-Response 三个属性，模板中可以通过 request、response、session 来引用，如 ${servlet.request.requestURL}。
- 所有 groupTemplate 的共享变量。

## 4.2.2　局部变量

在模板中定义的变量，只能在当前模板中使用，无法在子模板中使用：

```
@ var salary = user.salary*2;
<span>${user.name}</span>
```

Beetl 定义变量的方式与 JS 类似，比如：

```
@ var a=0,b="你好",d=true,e=34343.343434343434h,f = e+a;
@ var list = [1,2,3];
@ var map = {"name":"xiandafu","age":18,"data":list};
```

变量类型同 JavaScript 类型一样，上面的变量 e 是一个高精度数据，可以任意长，变量要以字母 h 结尾，以向 Beetl 表明这是高精度数据，对应了 Java 的 BigDecimal。在 Beetl 中，任何与高精度数据进行算数运算后的结果也必然转为 BigDecimal。

变量 map 和 list 有点像 JavaScript 中的 JSON，在 Beetl 中，分别是通过 HashMap 和 ArrayList 来实现的。

## 4.2.3　共享变量

类似全局变量，可以在任何模板中使用，需要通过 groupTemplate 的 API 添加。考虑为 JS 生成一个版本号，可以在 BeetlExtConfig 中添加如下代码：

```
@Configuration
public class BeetlExtConfig {
  @Autowired GroupTemplate groupTemplate;
```

```
    @PostConstruct
    public void config(){
      groupTemplate.getSharedVars().put("jsVersion",
System.currentTimeMillis());
    }
}
```

因此，可以在任何模板中使用 jsVersion 变量：

```
<script src="/js/xxxx.js?version=${jsVersion}" />
```

### 4.2.4 模板变量

变量的内容是模板对应的输出，比如：

```
@var a = 1;
@var template = {
  <span>${a}</span>
@};
输出：${template}
```

template 是个变量，内容位于{ }中，内容是 1，模板变量可以用在后面的任何地方，Beetl 使用模板变量来完成继承布局方式。

## 4.3 表达式

### 4.3.1 计算表达式

类似 JS，支持+、−、*、/、%等表达式，变量的类型与相应的 Java 类型一致：

```
@var b = 1;
@var a=(b+12)*3,c=12/5;
a=${a},b=${b}
```

a 的值是 39，c 的值是 2.4。计算中如果需要使用高精度数代替 double，则需要在数字后面加上“h”来表示这是一个高精度数据，对应了 Java 的 BigDecimal。

```
@var a =12323232323.232323h;
@var b= 343434.00001h;
<span>${a*b} ${a/b} </span>
```

此计算能保证浮点计算的准确性，计算过程中任何有高精度数的表达式都会导致整个过程按照高精度进行计算。

### 4.3.2 逻辑表达式

Beetl 支持类似 JavaScript、Java 的条件表达式，如>、<、==、!=、>= 、<= 、!、&&和 || ，以及三元表达式等，如以下例子所示。

```
@if(user.status==1&&user.salary<1000){
  <span>失业</span>
@}
```

三元表达式如果只考虑 true 条件对应的值的话，可以做简化，以下两行效果是一样的：

```
@ var a = 1 ;
${a==1?"selected":''}
${a==1?"selected"}
```

以上三元表达式中，a 的值为 1，因此表达式返回的结果是"selected"，如果 a 的值不是 1，则返回空字符串。在简化的三元表达实例中，则自动为 null，Beetl 占位符对 null 值不做输出，因此效果是一样的。

## 4.4 控制语句

### 4.4.1 循环语句

Beetl 支持多种循环方式，最常用的是 for…in 循环语句：

```
@for(user in userList){
<span>${user.name}</span>
@}
```

上面的例子中，for 循环会从 userList 中取出每一个元素赋值给 user 变量，可以在 for 的循环体中使用 user 变量。userList 可以是 Java 中的集合、数组。如果遍历的是 Java 中的 Map，则遍历的元素对应于 Map 的 Entry，因此，还需要通过 key 和 value 来获取遍历的元素，如：

```
@for(entry in configMap){
@var name = entry.key,user = entry.value;
 <span>${user.name}</span>
@}
```

for…in 循环支持 elsefor 语法，即如果未进入循环体，则执行 elsefor 部分，比如：

```
@ var array=[];
@for(item in array){
  ...
@}elsefor{
 <span>无数据</span>
@}
```

上面的例子中，array 变量定义为集合，因为是空集合，所以上述循环执行了 eslefor 部分。

for…in 支持在集合变量后加上感叹号以进行安全输出（参考 4.10 节安全输出的内容），如果变量不存在或者为 null，则不进入循环体，比如：

```
@ var data = null;
@ for(entry in data.userList!){


@}
```

在上面的例子中，因为变量 data 为空，所以 data.userList 会报出空指针错误，可以通过在变量表达式后面加上“!”来进行安全输出，意思是“我知道这个变量表达式可能不存在，也可能为空，不要报错”。在 for…in 循环中，自动解释为不报错，不进入循环体。

Beetl 循环还支持其他形式的循环，比如(for(exp;exp,exp)这种，或者 while(exp)这种。

```
@for(var i=1;i<10;i++){
  <span>${i}</span>
@}
```

> **注意：** for…in 循环也可以得到循环的上下文信息，在 for…in 循环中，会自动创建一个变量名+LP 后缀的变量，提供了循环的上下文信息，比如：
>
> ```
> @for(user in users){
>  ${userLP.index}
> @}
> ```
>
> 其他还有：
>
> - userLP.index，当前的索引，从 1 开始；
> - userLP.size，集合的长度；
> - userLP.first，是否是第一个；
> - userLP.last，是否是最后一个；
> - userLP.even，索引是否是偶数；
> - userLP.odd，索引是否是奇数。

## 4.4.2　条件语句

Beetl 的条件语句有 if else，同 Java 和 JS 的用法一样，还提供了 switch case 这样的语法，不同的是 switch case 支持任何类型的表达式。

```
@var a =true,b=1;
@if(a&&b==1){
<span> ...... </span>
@}else if(a){
<span> ...... </span>
@}else{
<span> ...... </span>
@}
```

同时 Beetl 也支持 switch case 的加强版 select-case，允许 case 中有逻辑表达式，同时，也不需要每个 case 都 break 一下，默认遇到符合条件的 case 执行后就退出（假设占位符设定为<%%>）。

```
<%
var b = 1;
select(b){
```

```
  case 0,1:
    print("it's small int");
  case 2,3:
    print("it's big int");
  default:
    print("error");
}
%>
```

case 中也支持逻辑表达式：

```
<%
var b = 1;
select{
  case b<1,b>10:
    print("it's out of range");
  default:
    print("error");
}
%>
```

## 4.4.3 try catch

模板渲染很少会遇到异常情况，但有些模板的高级应用仍然需要捕获异常，Beetl 也支持 try catch 结果来捕获异常：

```
<%
try{
  callOtherSystemView();
}catch(error){
  print("暂时无数据");
}
%>
```

error 代表了一个异常，可以通过 error.message 来获取可能的错误信息。

## 4.5 函数调用

Beetl 内置了大量的常用函数以辅助模板渲染、规则引擎等功能，函数调用方法与 JS 一样。当然也可以注册自定义的函数，自定义函数可以通过 Java 来实现，或者通过 Beetl 本身来实现。Beetl 提供的常用函数如下：

- print 和 println，输出对象，如果对象为空，则不输出。
- has，判断是否具有这个全局变量，如 if(has(userList)){....}，判断后台是否提供了 userList 变量。
- isEmpty，判断变量或者表达式是否为空，如果不存在，为 null，都返回 true；同时如果字符串空，集合空也返回 true；isNotEmpty 则相反。
- debug，在控制台打印变量或者表达式。此方法还会附带此调用所在的文件和模板，以及变量的名字等信息，方便追踪。
- date，日期函数，获得当前日期。
- trim，截取一个日期或者数字类型并返回字符串，如 trim(126.18,1)返回字符串 126.18，trim(date(),'yyyy')返回年份 2017。
- parseInt、parseLong、parseDouble，将字符串或者 number 转为对应的类型。
- global，返回一个全局变量值，参数是一个字符串，如 var user = global("user_"+i)。
- cookie，返回指定的 cookie 对象，如 var userCook = cookie("user"),allCookies = cookie()。
- strutil.*，系列函数，对字符串的操作，如 strutil.split("abc,def",",")将返回一个字符串数组，具体参考 Beetl 使用手册。
- array.*，集合的相关函数，如 array.contain。
- shiro.*，shiro 是常用的安全框架，并非是 Beetl 的内置函数（参考官网手册）。
- spring.*，Spring 框架中可以使用的一系列函数，比如 spel 表达式。
- reg.*，正则表达式的相关函数。

> **注意：**Beetl 中的函数，以及后面要讲的标签函数，都允许加上类似命名空间的命名规范，如 strutil.split，这个整体是函数的名字，这样的好处是便于管理庞大的函数。

## 4.6 格式化函数

允许在占位符输出的时候指定格式化函数来格式化输出，最常见的有日期格式化和数字格式化。这两种格式化与 Java 的 SimpleDateForamt 和 NumberFormat 一致。

格式化的格式是${exp,formatName="可选参数"}，比如格式化一个日期，可以用：

```
${date(),dateFormat="yyyy-MM-dd HH:mm"}
${date(),dateFormat}
${date(),"yyyy-MM-dd"}
```

以上三种形式都可以，第一种将使用 dateFormat 格式化函数，这是 Beetl 内置的格式化函数，且输出格式为 yyyy-MM-dd HH:mm。

第二种也使用了 dateFormat 格式化函数，采用系统默认的格式。

第三种没有指定是哪种格式化函数，因为 Beetl 内置了对日期类型使用 dateFormat 作为格式化函数。

数字格式化函数是 numberFormat，使用方式同 dateFormat，格式化方式与 java.text.DecimalFormat 一致。

## 4.7 直接调用 Java

可以在模板中以 Java 方式调用表达式，使用的时候，必须在表达式前使用@符号以表示其后表达式是 Java 风格的。

```
@var a = [1,2];// a 是 java.util.ArrayList
@var size = a.~size;
@var size2 = @a.size();
```

如以上代码所示，集合（数组）长度由 Beetl 提供的虚拟属性获取，虚拟属性有前缀~，表示其后面的属性并非对象本身的属性，而是虚拟的，可以通过注册为任何对象增加多个虚拟属性。

另外一种获取集合长度的方法就是调用集合本身的 size 方法，这里使用@a.size()告诉 Beetl：我要使用 Java 直接调用了。Java 的表达式是 a.size()，也就是求集合长度的方法。

可以调用实例的 public 方法和属性，也可以调用静态类的属性和方法，需要加一个@指示此调用是直接调用 class，其后的表达式是 Java 风格的，以下是更多的例子：

```
${@user.getMaxFriend("lucy")}
${@user.maxFriend[0].getName()}
${@com.xxxx.constants.Order.getMaxNum()}
${@com.xxxx.User$Gender.MAN}
<%
```

```
var max = @com.xxxx.constants.Order.MAX_NUM;
var c =1;
var d = @user.getAge(c);
%>
```

- 对于直接 Java 调用，groupTemplate 可以配置为禁止此功能，具体请参考 Beetl 官网文档。
- 也可以通过安全管理器配置到底 Beetl 不允许调用哪些类，默认情况下，java.lang.Runtime 和 java.lang.Process 不允许在模板里调用。你自己的安全管理器也可以配置为不能直接访问 DAO 类。
- 请按照 Java 规范书写类名、方法名和属性名。这样便于 Beetl 识别到底调用的是哪个类、哪个方法，否则会抛出错误。
- 可以省略包名，只用类名。Beetl 将搜索包路径找到合适的类（需要设置配置“IMPORT_PACKAGE=包名.;包名.”，包名后需要跟一个“.”。
- 内部类（包括枚举）访问同 Java 一样，比如 User 类有个内部枚举类 Gender，访问是 User$Gender。
- 表达式是 Java 风格的，但参数仍然是 Beetl 表达式，比如@user.sayHello(user.name)。这里的 user.sayHello 是 Java 调用，user.name 仍然是 Beetl 表达式。

## 4.8 标签函数

所谓标签函数，即允许处理模板文件中的一块内容，功能等同于 jsp tag。比如 Beetl 内置的 layout 标签 index.html，作为一个页面首页：

```
@layout("/inc/layout.html",{title:'主题'}){
<span>Hello,this is main part</span>
@}
```

layout 是一个布局的标签函数，它会将标签体的{ }部分的内容渲染出来后，传给 layout 指定的模板页面，其变量名默认为 layoutContent，同时也传递了 title 变量。

layout.html 是布局页面：

```
<title>${title}</title>
<div>
  ${layoutContent}
```

```
</div>
```

常用的标签函数还有 include，用来包含一个模板页面，我们将在 4.11.6 节详细说明。

## 4.9 HTML 标签

HTML 是一种特殊的标签函数，Beetl 在形式上与 HTML 标签类似，Beetl 默认通过#符号来区分 Beetl 的 HTML 标签。

```
<#footer style="simple"/>
<#richeditor id="rid" path="${ctxPath}/upload" name="rname"
maxlength="${maxlength}"> ${html} …其他模板内容
</#richdeitor>
<#html:input  id='aaaa' />
```

如上面的例子所示，<#表示这是一个 Beetl 的 HTML 标签，标签名字是 footer，标签属性是 style。为了实现这个 footer 标签，通常有两种实现方法，一种是用 Java 写一个标签函数，注册为 footer 的名字，我们将在高级部分讲述；另外一种是用 Beetl 语言本身实现，需要在 templates/htmltag 目录下创建一个 footer.tag 的文件，内容就是普通的模板内容。

```
@ if(style=='simple'){
请联系我 ${session.user.name}
@ }else{
请联系我 ${session.user.name},phone:${session.user.phone}
@ }
```

style 是标签中定义的变量，因此可以在 HTML 标签文件中使用。

以下是 HTML 标签函数的一些使用规则：

- 可以在自定义标签中引用标签体的内容，标签体可以是普通文本、Beetl 模板，以及嵌套的自定义标签等。上面的<#richeditor>标签体中，可用 tagBody 变量来引用。
- HTML 自定义标签的属性值均为字符串，如<#input value="123" />，在 input.tag 文件中变量 value 的类型是字符串。
- 可以在属性标签中引用 Beetl 变量，如<#input value="${user.age}" />，此时在 input.tag 中，value 的类型取决于 user.age。

- 在属性中引用 Beetl 变量，不支持格式化，如<#input value="${user.date,'yyyy-MM-dd' }" />，如果需要格式化，需要在 input.tag 文件中自行格式化。
- 在标签属性中传 JSON 变量需要谨慎，因为 JSON 包含了"}"，容易与占位符混合导致解析出错，因此得使用"\"符号，如<#input value="${ {*age*:25\} }" />。
- html tag 属性名将作为其对应模板的变量名，因此请确保命名符合变量名规范。

## 4.10 安全输出

模板语言和 Java 语言有一个很大的不同在于模板语言使用的全局变量有可能不存在，比如对应一个编辑用户的页面，修改功能有一个 user 对象，而新增功能，对于同样的这个模板，user 并不存在。

Beetl 在变量表达式后面使用符号"！"来提醒 Beetl 此变量可能不存在，表达式将返回"！"后的表达式值，如果其后没有表达式，则返回 null。

```
@ var user = null;
${user.wife.name!}
${user.wife.name!"单身汉"}
```

对于第一个表达式，使用了安全输出符号，因此当 user 为 null 时，或者 user 的 wife 属性为 null 时，其表达式返回值都是 null，第二个则返回表达式后面的值，其值可以是常量或者表达式。

> **注意：** Beetl 提供了 has 函数来判断全局变量是否存在，也可以使用 isEmpty 函数来判断全局变量是否不存在。
>
> ```
> @if(has(user)){
>  ....
> @}
> ```

## 4.11 高级功能

### 4.11.1 配置 Beetl

可以在 resources 下添加 beetl.properties 文件来个性化 Beetl，我们在 4.1.2 节已经通过此配

置文件配置了 Beetl 使用的定界符号和占位符号，此配置还包括更多的个性化配置，以下是常用的一些配置：

```
# 字节输出，默认是 false，改成 true 能提高 Web 的模板渲染性能
DIRECT_BYTE_OUTPUT = FALSE
# 是否允许 Java 调用，默认是允许
NATIVE_CALL = TRUE
# 模板的字符集
TEMPLATE_CHARSET = UTF-8
# 自动检查模板文件变化，默认是 true，开发模式下使用。如果是产品模式，最好改成 false 以
提高性能
RESOURCE.autoCheck= true
# 错误处理类，默认输出错误信息到控制台，包含错误的位置、错误的符号等信息
ERROR_HANDLER = org.beetl.core.ConsoleErrorHandler
# 自定义标签文件 Root 目录和后缀
RESOURCE.tagRoot = htmltag
RESOURCE.tagSuffix = tag
# 自定义脚本方法文件的 Root 目录和后缀
RESOURCE.functionRoot = functions
RESOURCE.functionSuffix = html
# HTML Tag 的符号
HTML_TAG_FLAG = #
# HTML Tag 中声明标签返回的变量的列表，具体参考 HTML 绑定变量标签
HTML_TAG_BINDING_ATTRIBUTE = var
## 内置的方法
FN.date = org.beetl.ext.fn.DateFunction
.....
## 内置的功能包
FNP.strutil = org.beetl.ext.fn.StringUtil
## 内置的格式化函数
FT.dateFormat =  org.beetl.ext.format.DateFormat
## 内置的默认格式化函数
FTC.java.util.Date = org.beetl.ext.format.DateFormat
.....
## 标签类
TAG.include= org.beetl.ext.tag.IncludeTag
```

## 4.11.2 自定义函数

自定义方法可以通过 Function 接口的 call 方法实现，call 方法有两个参数：

- Object[] paras，代表调用该方法的参数。
- Context ctx，调用方法的上下文参数，可以获取到
  - ByteWriter，Beetl 中的输出流；
  - globalVar，全局变量；
  - resourceId，当前正在渲染的模板的名称。

```
public class MyPrint implements Function{
  public String call(Object[] paras, Context ctx){
    Object o = paras[0];
    if (o != null){
      try{
        ctx.byteWriter.write(o.toString());
      }catch (IOException e){
        throw new RuntimeException(e);
      }
    }
    return "";
  }
}
```

以上方法实现了一个 Function 接口，用于打印第一个参数。可以在 beetl.properties 中注册这个 Beetl 函数：

```
FN.myPrint = xxx.MyPrint
```

FN 前缀表示这是一个函数配置项，myPrint 表示在 Beetl 中的名称，xxx.MyPrint 则是其实现类。Beetl 会在模板引擎初始化的时候创建此类的一个实例来服务所有的调用。如果你的应用中自定义的函数比较多，可以使用命名空间来管理，比如注册为：

```
FN.io.myPrint = xxx.MyPrint
```

这样，Beetl 中调用此函数的名字就是“io.myPrint”，比如：

```
@ io.myPrint("hello,my funcation");
```

也可以将普通类的所有public方法注册为自定义函数，称之为功能包，如果需要使用Context变量，只需要在 public 方法的最后一个参数中加上 Context 即可。比如你有一个类叫 IOUtil:

```
public class IOUtil{
  public Obect print(Object a, Context ctx){
    //
    return null;
  }
}
```

可以使用 FNP 来注册，该类的某个 public 方法都自动注册为 Beetl 扩展函数：

```
FNP.ioutil = xxx.IOUtil
```

这样，print 方法将自动注册为 ioutil.print 函数。

可以使用 Beetl 语言本身来实现函数，需要在 templates/functions 目录下添加扩展名为 html 的模板文件，方法参数分别对应 para0、para1……

以下是在 functions 目录下一个名为 print.html 的文件：

```
@ var obj = para0;
${obj}
@ return null;
```

会自动建立一个 print 方法。

> 有时候，自定义方法会使用 Spring 管理的 Bean，可以为自定义 Function 添加 @Component，使其成为 Spring 管理的 Bean。通过 4.1.4 节的方式来注册成为 groupTemplate 的扩展函数。

### 4.11.3 自定义格式化函数

需要实现 Format 接口或者 ContextFormat 接口，两个都一样，后者接口会提供 Context 参数，比如模板有可能输出一段 JS 脚本从而产生跨域脚本攻击（XXS），因此可以写一个格式化函数来处理这种情况。

```
import org.apache.commons.lang3.StringEscapeUtils;
public class XXSDefenderFormat implements Format{
 public Object format(Object data, String pattern){
  if(data instanceof String){
   String js = (String)data;
   String str = StringEscapeUtils.escapeHtml4(js);
    return str;
  }else{
    return data;
  }
 }
}
```

注册上述格式化函数，在 beetl.properties 中增加：

```
FT.xxs =  xxxx.XXSDefenderFormat
```

这样，可以在你的模板里使用 xxs 作为格式化函数：

```
${"<script>alert(1)</script>",xss}
```

## 4.11.4　自定义标签函数

标签函数类似 JSP Tag，运行处理一段模板内容，比如 layout 标签函数用于布局，还有后面要提到的 include 标签函数，HTML Tag 是一种特殊的标签函数，我们将在下一节再介绍。

标签函数是模板引擎里编写可重用的组件的一种方式。通过继承 Tag 类，并重载 render 方法来实现。

```
public class SimpleTag extends Tag{
 @Override
 public void render(){
  // do nothing,just ignore body
  ctx.byteWriter.write("被删除了，付费可以看")
 }
}
```

以上标签函数仅仅输出一个警告信息，调用了 Context 变量的 byteWriter。注册此标签函数需要在配置文件中使用 Tag 前缀：

```
TAG.myTag= xxx.SimpleTag
```

这样在模板中就可以使用 showTag 标签函数了，比如：

```
@myTag(){
 <span>这段内容，不会被看到 </span>
@}
```

Tag 类提供了如下方法：

- doBodyRender()，渲染标签片段并输出。
- getBodyContent()，渲染标签片段，得到渲染后的内容，返回的是一个 BodyContent 类，可以通过调用 getBody 方法获得其渲染的内容。
- 属性 args，标签传入的参数，是个数组列表，对应了传入的参数。
- ctx，是 Context，上下文信息，可以参考自定义函数以了解 Context。

> 有时候，自定义标签函数和后面讲的自定义 HTML 标签都会使用 Spring 管理的 Bean，可以为自定义 Tag 添加@Component，使其成为 Spring 管理的 Bean。通过 4.1.4 节的方式来注册成为 groupTemplate 的扩展标签函数，如下：

```
@Component
@Scope("prototype") // 每次使用都会创建 Tag，因此使用 prototype
public class SimpleTag extends Tag{
  .....
}

@PostConstruct
public void config(){
 // 注册一个标签
 groupTemplate.registerTagFactory("myTag", new TagFactory(){
    public Tag createTag() {
        return applicationContext.getBean(SimpleTag.class);
    }
 });
}
```

## 4.11.5 自定义 HTML 标签

除了在 4.11.4 节中提到的可以使用 Beetl 语言本身实现标签函数，也可以通过实现 Tag 类接口来实现 HTML 标签函数，此时 args 的第一个参数 args[0]表示标签名，通常没有什么用处，args[1]则是标签的属性，参数是个 map，key 是 HTML Tag 的属性，value 是其属性值，以下用 Java 完成的 HTML 标签用于输出属性值。

```
public class SimpleHtmlTag extends Tag{
  @Override
  public void render(){
    String tagName = (String) this.args[0];
    Map attrs = (Map) args[1];
    String value = (String) attrs.get("attr");
    try{
      this.ctx.byteWriter.writeString(value);
    }catch (IOException e){

    }
  }
}
```

如果注册此标签函数为 simpleTag，则可以在模板中使用这个标签函数：

```
<#simpleTag attr="hello"></#simpleTag>#
```

这个标签函数输出“hello”。

> **注意：** 配置 HTML_TAG_FLAG 默认为#，用来区别是否是 Beetl 的 HTML Tag，也可以设置成其他符号，比如设置了 HTML_TAG_FLAG=my:，那么，使用 simpleTag 应该是如下样子的：<my:simpleTag attr="hello"/>

有一类模板标签还需要标签实现返回一组变量供标签体渲染使用，通常称为绑定变量的 HTML Tag。比如在 CMS 系统中，可以定义 article 标签，该标签返回文章列表，标签体根据文章列表来进一步渲染成文件列表内容。

```
<#article type="1"  var="article,index">
  <div>
```

```
  ${index} ${artile.title,xss},${artile.postDate,'yyyy-MM-dd'}
  </div>
  </#article>
```

article 将循环显示 top10 新闻，其标签函数实现将从后台取出 top10 的文章，并循环调用标签体，在调用前，会将文章对象绑定到 article 变量，当前索引绑定到 index 变量中。

```
@Component
@Scope("prototype")
public class AtricleSample extends GeneralVarTagBinding{
  @Autowired XXXService service ;
  @Override
  public void render(){
    int type = Integer.parseInt((String) this.getAttributeValue("type"));
    List<Atricle> list = service.query(type);
    for (int i = 0; i < limit; i++){
      this.binds(list.get(i),i+1);
      this.doBodyRender();
    }
  }
}
```

带绑定变量的必须继承 GeneralVarTagBinding 类，此类提供了一个 binds 函数，定义如下：

```
public void binds(Object... array)
```

可以在 Java 代码中传入多个需要绑定的变量，与 var 属性中声明的一一对应上，否则，Beetl 会抛出错误。

## 4.11.6 布局

布局是保持模板重用的最重要的功能之一，Beetl 可以联合 include 和 layout 来实现重用，比如使用 include 标签函数：

```
@ include("/inc/header.html",{"title":"测试页面"}){}
<span>这是正文</span>
@ include("/inc/footer.html"){}
```

include 标签函数的第一个参数是公共模板的路径，其后可以接受一个 map 参数，map 中的每一项值都会传递给子模板作为子模板的全局变量。

如果整个模板风格固定，也可以使用 layout 标签函数来实现。layout 指定了一个布局页面，因此模板渲染后的结果将回填到布局页面中：

```
@ layout("/inc/layout.html",{"title":"测试页面"}){
  <span>这是正文</span>
@ }
```

layout.html 是一个布局页面，隐含地使用了变量 layoutConent 来代表标签体渲染的内容，因此 layout.html 的内容如下：

```
<title>${title}</title>
<body>
  ${layoutContent}
</body>
```

还有一种比 layout 布局更为复杂的布局方式，称为继承布局，即父页面定义了页面的大致结构，但需要子页面回填多个内容片段。比如父页面需要回填正文，还要回填 JS 引入。这种布局混合使用了 include 和模板变量。定义父页面的内容如下：

```
</head>
<script src=""/>
${jsPart}
<head>
<body>
${bodyPart}
</body>
```

子页面可以使用模板变量来完成，内容如下：

```
@var jsPart={
<script src=""/>
<script src=""/>
@};

@var bodyPart={
```

```
    <span></span>
@};

@include("/inc/layout.html",{"jsPart":jsPart,"bodyPart":bodyPart}){}
```

子页面定义了两个模板变量，分别是 jsPart 和 bodyPart，模板变量的内容是模板渲染结果，因此这两个变量就包含了 layout.html 页面需要的部分。

> **注意：** 实现模板继承的方法很多，但都基于以上类似模板变量+include 的实现方式，如果熟悉了 Beetl 的 HTML Tag，完全可以实现如下的继承方式（这个可以参考 Beetl 官网来实现）：
>
> ```
> <#header>
>
> </#header>
> <#body>
> </#body>
> <#extends file="/inc/layout.html"/>
> ```

## 4.11.7 AJAX 局部渲染

可以用 AJAX 标记标注模板的一个片段，在 Controller 中可以仅渲染这个片段而不是整个模板，常常应用在 AJAX 请求中，如翻页等 AJAX 功能。以下是一个 user.btl 模板：

```
<div id ="pageContainer">
@ #ajax userPage:{
<table>
  @for(user in userList){

  @}
</table>
@}
</div>
```

上面的模板页面使用#ajax 标注了一段 AJAX 模板。正常渲染整个模板的时候，语法#ajax 被忽略，内容正常处理，跟没有#ajax 一样。如果 Controller 中声明只渲染 user.btl#userPage，则 Beetl 模板引擎只会渲染#ajax userPage 部分：

```
@RequestMapping("/user/queryUser.html")
public ModelAndView queryUser(){
  ModelAndView view = new ModeAndView();
  view.setViewName("/user/user.btl#userPage");
  view.addObject("userList",uses);
  return view;
}
```

这样，渲染结果仅仅是 table 部分的内容，前端开发者可以将此渲染结果回填到 id 为 pageContainer 的部分，从而实现 AJAX 加载模板片段，比如，使用 jQuery：

```
$("#pageContainer").load("/user/queryUser.html",params);
```

## 4.12 脚本引擎

Beetl 是模板引擎，但本质上还是一个脚本语言，它在解析阶段会将模板转化为 Beetl 脚本执行。groupTemplate 提供了执行脚本的功能，并能获取到返回值。

Beetl 的脚本功能可以作为业务规则引擎和脚本引擎的基础，groupTemplate 提供如下 API：

- public Map runScript(String key, Map paras) throws ScriptEvalError；
- public Map runScript(String key, Map paras, Writer w) throws ScriptEvalError；
- public Map runScript(String key, Map paras, Writer w, ResourceLoader loader) throws ScriptEvalError；
- public BeetlException validateTemplate(String key)，校验脚本，如果出错，返回一个异常类，BeetlException 包含了异常的详细信息。

比如执行一个脚本“return 1300+月薪;”：

```
  String script = " return 1300+月薪;";
  // 资源加载器，实例的资源是字符串，而不是通常的模板文件，因此使用 StringTemplate-
ResourceLoader
  StringTemplateResourceLoader tempLoader = new
StringTemplateResourceLoader();
  Map<String,Object> paras = new HashMap<String,Object>();
  paras.put("月薪",5100);
  Map ret = groupTemplate.runScript(script,paras,tempLoader);
```

```
// Integer 类型
Number num = (Number)ret.get("return");
```

runScript 方法总会返回一个 Map，包含 return 的变量，也包含了模板里所有定义的顶级变量，如以下脚本：

```
var 权重 = getDBConfig("权重");
var ret = 权重*1.2;
```

执行此脚本后，返回的 Map 中包含参数“权重”和“ret”。

## 4.13 JSON 技术

在 MVC 框架中，Spring Boot 内置了 Jackson 来完成 JSON 的序列化和反序列化操作。而且，在与其他技术集成的时候，比如 Redis、MongoDB、Elasticsearch 等对象序列化，默认都是使用 Jackson 来完成的，因此关于 JSON 技术，本书主要介绍 Jackson 工具。国内阿里巴巴提供的 Fastjson 也是个很好的工具，由于篇幅有限，这里不再介绍了。

总的来说，JackSon 功能强大，既能满足简单的序列化和反序列化操作，也能实现复杂的、个性化的序列化和反序列化操作。到目前为止，Jackson 的序列化和反序列化性能都非常优秀，已经是国内外大部分 JSON 相关编程的首选工具。

### 4.13.1 在 Spring Boot 中使用 Jackson

在 Controller 中，方法注解为@ResponseBody，则自动将方法返回的对象序列化成 JSON。

```
@Controller
@RequestMapping("/json")
public class BeetlController {
  @Autowired UserService userService;
  @GetMapping("/user/{id}.json")
  public @ResponseBody User showUserInfo( @PathVariable Long id){
    User user = userService.getUserById(id);
    return user;
  }
}
```

## 4.13.2 自定义 ObjectMapper

如果想自定义一个 ObjectMapper 来代替默认的，可以使用 Java Config，使用@Bean 来配置一个，代码如下：

```
@Configuration
public class JacksonConf {
  @Bean
  public ObjectMapper getObjectMapper() {
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    return objectMapper;
  }
}
```

以上 Java Config 令 Spring Boot 使用自定义的 Jackson 来序列化而非默认配置的。以下是一个用来获取当前时间的请求：

```
@GetMapping("/now.json")
public @ResponseBody Map datetime(){
  Map map = new HashMap();
  map.put("time", new Date());
  return map;
}
```

当用户访问 now.json 的时候，会得到如下输出：

```
{"time":"2017-04-12 22:52:05"}
```

序列化 JSON，有时候并非如上面的例子那么简单，当需要忽略某些字段，或者有些字段要按照特定格式进行序列化时，我们使用 Jackson 可以较为容易地做到，我们将在下面各小节中详细讲解。

## 4.13.3 Jackson 的三种使用方式

Jackson 是一个流行的高性能 JavaBean 到 JSON 的绑定工具，Jackson 使用 ObjectMapper 类将 POJO 对象序列化成 JSON 字符串，也能将 JSON 字符串反序列化成 POJO 对象。实际上，

JackSon 支持三种层次的序列化和反序列化方式。

- 采用 JsonParser 来解析 JSON，解析结果是一串 Tokens，采用 JsonGenerator 来生成 JSON，这是最底层的方式。
- 采用树遍历（Tree Traversing）方式，JSON 被读入到 JsonNode 对象中，可以像操作 XML DOM 那样读取 JSON。
- 采用 DataBind 方式，将 POJO 序列化成 JSON，或者反序列化到 POJO，这是最直接和最简单的一种方式，不过有时候需要辅助 Jackson 的注解或者上述序列化实现类来个性化序列化和反序列化操作。

对于应用程序来说，最常用的方式是 DataBind，也就是将 POJO 对象转化为 JSON 字符串，或者解析 JSON 字符串并映射到 POJO 对象上。树遍历也是一种常用方式，特别是没有现成的 POJO 做数据绑定的时候，可以遍历树来获取 JSON 数据。

最底层的 Parser 和 Generator 在应用程序中经常辅助用于个性化序列化和反序列化，我们将在说明@JsonSerialize 的时候详细讲解。

## 4.13.4 Jackson 树遍历

树遍历方式通常适合没有 POJO 对应的 JSON，代码如下：

```
@Autowired ObjectMapper mapper;

@GetMapping("/readtree.json")
public @ResponseBody String readtree() throws JsonProcessingException,
IOException{
  String json = "{\"name\":\"lijz\",\"id\":10}";
  JsonNode node = mapper.readTree(json);
  String name = node.get("name").asText();
  int id = node.get("id").asInt();
  return "name:"+name+",id:"+id;
}
```

readTree 方法可以接受一个字符串或者字节数组、文件、InputStream 等，返回 JsonNode 作为根节点，你可以像操作 XML DOM 那样操作遍历 JsonNode 以获取数据。

JsonNode 支持以下方法来读取 JSON 数据：

- asXXX，比如 asText、asBoolean、asInt 等，读取 JsonNode 对应的值。

- isArray，用于判断 JsonNode 是否是数组，如果是数组，则可以调用 get(i)来进行遍历，通过 size()来获取长度。
- get(String)，获取当前节点的子节点，返回 JsonNode，如以上代码所示。

> **注意：** JSON 规范要求 Key 是字符串，且用双引号，尽管很多工具都可以用单引号甚至不用也能识别，但还是建议遵照 JSON 的规范。

## 4.13.5 对象绑定

应用程序更常见的是使用 Java 对象来与 JSON 数据互相绑定，仅仅调用 ObjectMapper 的 readValue 来实现，比如在上一个例子中，JSON 如下：

```
{"name":"lijz",age:10}
```

可以创建一个 POJO 对象来与 JSON 相对应，对象如下：

```
public class User {
    Long id;
    String name;
    // 忽略 getter 和 setter 方法
}
```

然后使用 readValue 来反序列化上面的 JSON 字符串：

```
@GetMapping("/databind.json")
public @ResponseBody String databind() throws JsonProcessingException,
IOException{
  String json = "{\"name\":\"lijz\",\"id\":10}";
  User user = mapper.readValue(json, User.class);
  return "name:"+user.getName()+",id:"+user.getId();
}
```

将 POJO 序列化成 JSON，使用 mapper 的 writeValueAsString 方法：

```
@GetMapping("/serialization.json")
public @ResponseBody String custom() throws JsonProcessingException {
  User user= new User();
```

```
user.setId(1l);
user.setName("hello");
String str = mapper.writeValueAsString(user);
return str;
}
```

mapper.writeValueAsString 将对象序列化成 JSON 字符串，可以使用 Jackson 注解来对序列化的字段进行定制。

### 4.13.6 流式操作

树模型和数据绑定都是基于流式操作完成的，即通过 JsonParser 类解析 JSON，形成 JsonToken 流，下面的代码是解析 JSON：

```
@Autowired
    ObjectMapper mapper;

    @RequestMapping("/parser.html")
    public @ResponseBody String parser() throws JsonParseException,
IOException{
        String json = "{\"name\":\"lijz\",\"id\":10}";
        JsonFactory f = mapper.getFactory();
        String key=null,value=null;
        JsonParser parser = f.createParser(json);
        // {, START_OBJECT,忽略第一个 Token
        JsonToken token = parser.nextToken();
        //"name",FIELD_NAME
        token = parser.nextToken();
        if(token==JsonToken.FIELD_NAME){
          key = parser.getCurrentName();
        }
        token = parser.nextToken();
        //"lijz", VALUE_STRING
        value = parser.getValueAsString();
        parser.close();
        return key+","+value;
    }
```

JsonParser 的解析结果包含了一系列 JsonToken，JsonToken 是一个枚举类型，常用的 START_OBJECT 代表符号"{"; START_ARRAY 和 END_ARRAY 代表"["和"]", FIELD_NAME 表示一个 JSON Key；VALUE_STRING 代表一个 JSON Value，字符串类型；VALUE_NUMBER_INT 则表示一个整数类型。

判断 Token 的类型后，通过调用 getValueAsXXX 来获取其值，XXX 是其值的类型。

```
@RequestMapping("/generator.html")
public @ResponseBody String generator() throws JsonParseException,
IOException{
  JsonFactory f = mapper.getFactory();
  // 输出到 stringWriter
  StringWriter sw = new StringWriter();
  JsonGenerator g = f.createGenerator(sw);
  // {
  g.writeStartObject();
  // "message", "Hello world!"
  g.writeStringField("name", "lijiazhi");
  // }
  g.writeEndObject();
  g.close();
  return sw.toString();
}
```

## 4.13.7 Jackson 注解

Jackson 包含了很多注解，用来个性化序列化和反序列化操作，主要有如下注解。

@JsonProperty，作用在属性上，用来为 JSON Key 指定一个别名。

```
@JsonProperty ("userName")
private String name
```

@JsonIgnore，作用在属性上，用来忽略此属性。

```
@JsonIgnore
private String password
```

@JsonIgnoreProperties，忽略一组属性，作用于类上，比如@JsonIgnoreProperties({ "id","photo" })。

```
@JsonIgnoreProperties ({"id","photo"})
public static class SamplePojo{
}
```

@JsonAnySetter，标记在某个方法上，此方法接受 Key、Value 两个参数，用于 JackSon 在反序列化过程中，未找到的对应属性都调用此方法。通常这个方法用一个 map 来实现：

```
@JsonAnySetter
private void other( String property, Object value ) {
  map.put(property,value);
}
```

@JsonAnyGetter，此注解标注在一个返回 Map 的方法上，Jackson 会取出 Map 中的每一个值进行序列化。

```
class Department {
        Map map = new HashMap();
        int id ;
        public Department(int id){
            this.id = id;
            map.put("newAttr", 1);
        }
        @JsonAnyGetter
        public Map<String, Object> getOtherProperties() {
          return map;
        }
}
```

Department，对象序列化的时候，其 JSON 类似：

```
{"id":1,"newAttr":1}
```

@JsonFormat，用于日期格式化，如：

```
@JsonFormat(pattern = "yyyy-MM-dd HH-mm-ss")
private Date createDate;
```

@JsonNaming，用于指定一个命名策略，作用于类或者属性上，类似@JsonProperty，但是自动命名。Jackson 自带了多种命名策略，你可以实现自己的命名策略，比如输出的 key 由 Java 命名方式转为下画线命名方法——userName 转化为 user-name。

```
@JsonNaming(PropertyNamingStrategy.LowerCaseWithUnderscoresStrategy.class)
public class Message {
    ...
}
```

LowerCaseWithUnderscoresStrategy，将所有属性名从驼峰命名方式转为下画线方式。

@JsonSerialize，指定一个实现类来自定义序列化。类必须实现 JsonSerializer 接口，代码如下：

```
public static class Usererializer extends JsonSerializer<User> {
  @Override
  public void serialize(User value, JsonGenerator jgen, SerializerProvider
provider)
    throws IOException, JsonProcessingException {
      jgen.writeStartObject();
      jgen.writeStringField("user-name", value.getName());
      jgen.writeEndObject();
  }
}
```

JsonGenerator 对象是 JackSon 底层的序列化实现，上面的代码中我们仅仅序列化 name 属性，且输出的 key 是 user-name。

使用注解 JsonSerialize 来指定 User 对象的序列化方式：

```
@JsonSerialize(using = Usererializer.class)
public class User {
  ...
}
```

@JsonDeserialize，用户自定义反序列化，同 JsonSerialize，类需要实现 JsonDeserializer 接口。

```
public class UserDeserializer extends JsonDeserializer<User> {
```

```
    @Override
    public User deserialize(JsonParser jp, DeserializationContext ctxt)
      throws IOException, JsonProcessingException {
        JsonNode node = jp.getCodec().readTree(jp);
        String name = node.get("user-name").asText();
        User user = new User();
        user.setName(name);
        return user;
    }
}
```

使用注解 JsonDeserialize 来指定 User 对象的序列化方式：

```
@JsonDeserialize (using = UserDeserializer.class)
public class User {
    ...
}
```

@JsonView，作用在类或者属性上，用来定义一个序列化组。Spring MVC 的 Controller 方法可以使用同样的@JsonView 来序列化属于这一组的配置。

比如对于 User 对象，某些情况下只返回 id 属性就行，而某些情况下需要返回 id 和名称。因此 User 对象可以定义成如下：

```
public class User {
    public interface IdView {};
    public interface IdNameView extends IdView {};

    @JsonView(IdView.class)
    private Integer id;
    @JsonView(IdNameView.class)
    private String name;

    // 忽略其他属性

}
```

User 定义了两个接口类，一个为 IdView，另外一个为 IdNameView，继承了 IdView 接口。这两个接口代表了两个序列化组的名称。属性 id 使用了@JsonView(IdView.class)，而属性 name

使用了@JsonView(IdNameView.class)。

Spring MVC 的 Controller 方法允许使用@JsonView()指定一个组名，被序列化的对象只有在这个组的属性才会被序列化，代码如下：

```
@JsonView(User.IdView.class)
@RequestMapping("/id.json")
public @ResponseBody User queryIds() {
  User user = new User();
  user.setId(1);
  user.setName("hell");
  return user;
}
```

以上代码将只输出 id 属性。如果换成@JsonView(User.IdNameView.class)，则输出 name 属性。同时，id 属性的组名 IdView.class 因为是 IdNameView.class 的父类，同样也会输出。

Jackson 除了以上常用注解，还包含了大量注解，需要参考官网 https://github.com/FasterXML/jackson-docs/wiki/JacksonAnnotations 来获取详细信息。

## 4.13.8 集合的反序列化

在 Spring MVC 的 Controller 方法中，可以使用@RequestBody 将提交的 JSON 自动映射到方法参数上，比如：

```
@RequestMapping("/updateUsers.json")
public @ResponseBody String say(@RequestBody List<User> list){
  StringBuilder sb = new StringBuilder();
  for(User user:list){
    sb.append(user.getName()).append(" ");
  }
  return sb.toString();
}
```

可以接受如下一个 JSON 请求，并自动映射到 User 对象上：

```
[
  {
```

```
    "name":"hello",
    "id":1
  },
  {
    "name":"world",
    "id":2
  }
]
```

Spring Boot 能自动识别出 List 对象包含的是 User 类，因为在方法中定义的泛型的类型会被保留在字节码中，所以 Spring Boot 能识别 List 包含的泛型类型从而能正确反序列化。

有些情况下，集合对象并没有包含泛型定义，如下代码所示，反序列化并不能得到期望的结果。

```
@RequestMapping("/customize.json")
public @ResponseBody String customize() throws Exception{
  String jsonInput = "[{\"id\":2,\"name\":\"xiandafu\"},{\"id\":3,\"name\":
\"lucy\"}]";
  // 错误的序列化
  List<User> list = mapper.readValue(jsonInput, List.class);
  return String.valueOf(list.size());
}
```

List 对象中的元素并非是 User，而是包含了一个 key 是 id 和 name 的 Map，这是因为 Jackson 并不知道要把 jsonInput 反序列化成 User 对象。在运行时刻，泛型已经被擦除了（不同于方法参数定义的泛型，不会被擦除）。为了提供泛型信息，Jackson 提供了 JavaType，用来指明集合类型，比如应用可以提供一个通用 getCollectionType：

```
public JavaType getCollectionType(Class<?> collectionClass, Class<?>...
elementClasses) {
  return mapper.getTypeFactory().constructParametricType(collectionClass,
elementClasses);
}
```

上面的反序列化代码可以改成如下内容：

```
JavaType type = getCollectionType(List.class,User.class);
```

```
List<User> list = mapper.readValue(jsonInput, type);
```

constructParametricType 方法允许构造复杂的泛型类型描述。

- List<Set,User>：使用 constructParametricType(List.class,Set.class,User.class)；
- Map<String,User>：使用 constructParametricType(Map.class,String.class,User.class)。

## 4.14 MVC 分离开发

在前端技术人员充足的公司里，越来越流行 MVC 分离开发，前端团队承担页面开发，后端团队只负责提供数据。Beetl 支持这种分离开发，能让前端人员以自己熟悉的方式模拟后端提供的数据，无论是 JSON 数据，还是模板需要的渲染数据，都可以在 Beet 的支持下完成。

Beetl 提供了 WebSimulate 来支持这种分离。前端人员只需要编写 Beetl 脚本，模拟返回值即可。而 Beetl 因为语言和使用习俗更接近 JavaScript，所以完成后端数据模拟非常容易。

### 4.14.1 集成 WebSimulate

如果使用 beetl-framework-starter，WebSimulate 实例已经配置好，可以直接使用，编写一个 Controller 来启用 WebSimulate：

```
@Controller
public class SimulateController {
  @Autowired
  WebSimulate webSimulate;

  @RequestMapping("/api/**/*.json")
  public void simluateJson(HttpServletRequest request,HttpServletResponse
response){
    webSimulate.execute(request, response);
  }
}
```

上面的这个例子模拟所有 API 开头的 JSON 请求。这里 RequestMapping 采用了“**”作为通配符号，可以匹配任何以 API 开头但还未完成的 Controller，都将交给 WebSimulate 暂时模拟实现。

## 4.14.2 模拟 JSON 响应

WebSimulate的工作目录默认是template/values目录，对应于Controller请求/api/user/1.json，WebSimulate 会执行 /values/api/user/1.json.var 脚本文件，并期望此脚本文件包含一个叫 json 的变量，比如

```
/* 位于/values/api/user/1.json.var */
var name = "xiandafu";
var json = {"name":name};
```

当浏览器调用/api/user/1.json 时，将得到一个模拟的 JSON。

WebSimulate 会对请求的 URI 路径做一定转化，以匹配 values 目录下的某个脚本，匹配规则如下：

- 完全匹配，访问/user/1.json，将匹配/user/1.json.var 脚本；
- 匹配 HTTP Method，访问/user/1.json，如果 HTTP Method 是 GET，则会优先匹配/user/1.json.get.var；
- 可以使用$$进行路径区配，通过 GET 访问/user/1.json，可以匹配/user/$$.var 或者/user/$$.get.var，匹配的值可以从全局变量 pathVars 中获取。比如，在/user/$$.var 文件中：

```
var userId = parseInt(pathVars[0]);
var json = {"name":name,"id":userId};
```

## 4.14.3 模拟模板渲染

可以在 SimulateController 方法中增加对模板渲染的模拟：

```
@RequestMapping("/**/*.html")
public void simluateJson(HttpServletRequest request,HttpServletResponse
response){
  webSimulate.execute(request, response);
}
```

我们假定所有 html 结尾的请求都交给 WebSimulate 来处理，同模拟 JSON 一样，由于采用了“**”作为通配符号，优先级低，因此一旦你的 Controller 代码准备好了，自动会切换到非模拟代码。

模拟模板渲染期望脚本能渲染所有的变量和视图名称。默认配置下，脚本中的所有变量都是视图的全局变量，脚本里称为 view 的变量指明了视图的名称。如果没有 view 变量，则认为 URL 的文件名就是模板名。

比如访问/user/userlist.html，WebSimulate 会先调用/values/user/userlist.html.var，获取到渲染的模板需要的数据，如果/values/userlist.html.var 脚本中包含了 view 变量，则会渲染此变量对应的视图。如果不存在，则默认渲染/user/userlist.html。

```
/* 位于/values/user/userlist.html.var */
var users = [{name:"xiandafu"},{name:"lucy"}];
var view = "/user/userlist.btl";
```

WebSimulate 会执行上面的脚本，构造类似 Spring MVC 的 ModelAndView，视图名称是“/user/userlist.btl”，用到的全局变量就是脚本定义的所有顶级变量。

在/values 目录下，允许将一些公共变量放到 common.var 中，比如 session 数据：

```
var session = {user:{name:"xiandafu"}};
```

这样，在模拟 JSON 或者模板渲染时，可以直接使用 session 变量。

> 由于篇幅有限，关于 WebSimulate 使用和定制请参考 Beetl 官网：ibeetl.com。

# 第 5 章 数据库访问

本章介绍 Spring JDBC Template 和 BeetlSQL 两种数据库访问方式，JDBC Template 是 Spring 自带的，在 JDBC 的基础上做了一定封装，而 BeetlSQL 是笔者研发的，除了封装了 JDBC 操作，还带有 SQL 管理、跨数据库平台支持等企业功能。它们的共同点都是以 SQL 为核心。下一章要介绍的 Spring Data，则是以对象为核心访问数据库的方式。

以 SQL 为中心或者以对象为中心，两种访问方式没有绝对的好坏，一般来说，以 SQL 为核心的数据库访问更为灵活，更能适应大型的互联网和企业应用，学习门槛较低。以对象方式访问数据库更适合较为简单的系统或者工具类系统，学习门槛刚开始较低，但因为 ORM（Object Relational Mapping）持久化上下文等概念过于复杂，导致后期深入学习较为困难，关于以对象为中心的方式，我们将在第 6 章 Spring Data JPA 中详细讲述。

## 5.1 配置数据源

本章将使用号称最快的数据库连接池 HikariCP 来说明 JDBC Template 和 BeetlSQL，集成 HikariCP 到 Spring Boot，首先在 pom 中加入如下依赖：

```
<dependency>
  <groupId>com.zaxxer</groupId>
  <artifactId>HikariCP</artifactId>
  <version>2.6.1</version>
```

```
</dependency>
```

截止到本书交稿的时候，HikariCP 的版本是 2.6.1，读者可以用最新的版本来代替本例中的版本。

然后在配置文件中配置连接数据库的基本信息以供 HikariCP 使用：

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/orm?
useUnicode=true&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

> **注意**：本例使用了 MySQL，你也可以使用任何其他驱动来完成本章的例子，使用 MySQL 需要在 pom 中配置 MySQL 驱动依赖。
>
> ```
> <dependency>
>  <groupId>mysql</groupId>
>  <artifactId>mysql-connector-java</artifactId>
>  <version>6.0.5</version>
> </dependency>
> ```

配置好上述依赖包和配置文件后，需要写一个 Java Config 来创建一个数据源，代码如下：

```
@Configuration
public class DataSourceConfig {
  @Bean(name = "dataSource")
  public DataSource datasource(Environment env) {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl(env.getProperty("spring.datasource.url"));
    ds.setUsername(env.getProperty("spring.datasource.username"));
    ds.setPassword(env.getProperty("spring.datasource.password"));
    ds.setDriverClassName(env.getProperty("spring.datasource.driver-
class-name"));
    return ds;
  }
}
```

以上代码创建了一个叫 dataSource 的数据源，我们使用 HikariDataSource。

> Environment 类在 Spring Boot 中代表了环境上下文，包含了 application.properties 配置属性、JVM 系统属性和操作系统环境变量的类，可以参考第 7 章了解 Environment。

为了快速尝试，需要准备一个 MySQL 数据库或者其他任何数据库，然后执行如下 SQL 脚本：

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) COLLATE utf8_bin DEFAULT NULL COMMENT '名称',
  `department_id` int(11) DEFAULT NULL,
  `create_time` date DEFAULT NULL COMMENT '创建时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;

CREATE TABLE `department` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) COLLATE utf8_bin DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

> 本章 JDBC Template 和 BeetlSQL 的例子及下一章的 Spring Data JPA 都会使用到上面的两张表。

## 5.2 Spring JDBC Template

Spring 提供了 JdbcTemplate 对数据库访问技术 JDBC 做了一定封装，包括管理数据库连接，简单查询结果映射成 Java 对象，复杂的结果集通过实现 RowMapper 接口来映射到 Java 对象。

在 Spring Boot 中，只要配置好数据源 DataSource，就能自动使用 JdbcTemplate。

```
@Repository
public class UserDao  {
    @Autowired
    JdbcTemplate jdbcTempalte;
}
```

@Repository 是 Spring 提供的注解，作用于类上，同@Service、@Configuration 一样，用于

引起 Spring 容器的注意。@Repository 通常用在同存储相关的类上。

## 5.2.1 查询

一个简单的返回总数的查询：

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from user",
Integer.class);
```

一个带参数绑定的查询：

```
Strign rowCount = this.jdbcTemplate.queryForObject("select count(*) from
user where department_id=? ", Integer.class,1);
```

如果期望返回 POJO 实例，JdbcTemplate 需要一个 RowMapper，将查询结果集 ResultSet 映射成一个对象：

```
public User  getUser(Long userId){
  String sql = "select * from user where user_id=?";
  User user = jdbcTempalte.queryForObject(sql, new RowMapper<User>() {
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
      User user = new User();
      user.setId(rs.getInt("id"));
      user.setName(rs.getString("name"));
      user.setDepartmentId(rs.getInt("department_ud"));

      return user;
    }
  }, userId);
  return user;
}
```

通常 RowMapper 可以被其他查询复用，因此最好的办法是在 DAO 中创建一个内部类，然后在 user 的查询中使用此类：

```
public class UserDao  {
```

```
  static class UserRowMapper implements RowMapper<User>{
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
      User user = new User();
      user.setId(rs.getInt("id"));
      user.setName(rs.getString("name"));
      user.setDepartmentId(rs.getInt("department_ud"));
      return user;
    }

  }

}
```

JdbcTemplate 查询如果期望返回的是列表，则需要使用 query 方法：

```
public List<User> getUserByDepartmentId(Long departmenetId){
  String sql = "select * from user w where department_id=?";
  List<User> user = jdbcTempalte.query(sql, new UserRowMapper(), departmenetId);
  return user;
}
```

> 对于大多数 DAO 框架来说，完成上述的查询会非常简单，不需要写 SQL，不需要写 RowMapper，比如后面要介绍的 BeetlSQL，以及 Spring Data 都是这种情况。以 BeetlSQL 为例，通过 sqlManager 可以直接完成上面的两个例子：
>
> ```
> // 根据主键查询
> User user = sqlManager.unique(User.class,userId);
> // 查询部门的所有用户
> User template = new User();
> template.setDepartmentId(departmentId);
> List<User> list = sqlManager.template(template);
> ```
>
> 大多数 DAO 框架都会自动生成 SQL，以及完成 ResultSet 到 POJO 的自动映射。

JdbcTemplate 允许查询结果返回一个 Map 而不是 POJO，这样免去了 RowMapper 的工作，数据库的字段名就是 Map 的 key，代码如下：

```
String sql = "select * from user where id=?";
```

```
Map map = jdbcTempalte.queryForMap(sql, userId);
```

如果输出 map 到控制台，则能看到如下代码：

```
{id=163, name=NewName, department_id=163, create_time=null}
```

> **注意**：使用 Map 作为查询结果有非常多的弊端，最严重的弊端是 Map 本身不适合程序的阅读，代码阅读者（可能是开发者本身、项目管理人员、代码后期维护者）很难通过 Map 了解查询结果集，只能通过阅读 SQL 代码或者阅读数据库定义来了解。另外严重的是，返回结果类型未知，如上面的例子中 id 是 Integer、Long，还是 BigDecimal，也只有在运行的时候才知道。还有，因为数据库不一样，同样的 id，有可能返回不同的数据类型，导致数据库平台不兼容。

## 5.2.2 修改

JdbcTempalte 提供 update 方法来实现 SQL 的修改语句，包括新增、修改、删除、执行存储过程等。

```
  String sql = "update user set name=? and departmet_id=? where id = ?";
  jdbcTempalte.update(sql,
user.getName(),user.getDepartmentId(),user.getId());
```

如果是数据库插入，操作同上面的 update，对于 MySQL、SQL Server 等数据库，含有自增序列，则需要提供一个 KeyHolder 来放置返回的序列：

```
  public Integer insertUser(final User user) {
    final String sql = "insert into user (name, departmet_id ) values (?,?)";
    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcTempalte.update(new PreparedStatementCreator() {
      public PreparedStatement createPreparedStatement(Connection connection)
throws SQLException {
        // 指出自增主键的列名
        PreparedStatement ps = connection.prepareStatement(sql, new String[]
{ "id" });
        ps.setString(1, user.getName());
        ps.setInt(2, user.getDepartmentId());
        return ps;
```

```
      }
    }, keyHolder);
    return keyHolder.getKey().intValue();
  }
```

上面的代码有点类似传统的 JDBC 访问，先声明一个 PreparedStatement，传入 SQL，以及自增长主键的列名。

```
PreparedStatement ps = connection.prepareStatement(sql, new String[]
{ "id" });
```

keyHolder 包含了自增长的结果，正如前面所说的，这并不能确定其序列类型，因此需要根据业务要求转化为 int 或者 long 类型。

> 对于大多数封装更好的工具来说，这段代码都会被 DAO 工具极大简化，以 BeetlSQL 为例，上述代码用一行即可完成：
>
> sqlManager.insert(user,true);
>
> 第二个参数为 true，表示需要获取自增序列，BeetlSQL 会自动获取到自增主键并赋值给属性 id。

## 5.2.3 JdbcTemplate 增强

NamedParameterJdbcTemplate 继承了 JdbcTemplate，不同于 JdbcTemplate，对 SQL 中的参数只支持传统的“?”占位符。NamedParameterJdbcTemplate 允许 SQL 中使用参数的名字作为占位符。

Spring Boot 已经自动配置了 NamedParameterJdbcTemplate，可以自动注入使用：

```
@Autowired
NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

5.2.1 节的查询部门用户总数的例子可以改成如下方式：

```
String sql = "select count(1) from user where department_id=:deptId";
MapSqlParameterSource namedParameters = new MapSqlParameterSource();
namedParameters.addValue("deptId", departmentId);
```

```
Integer count = namedParameterJdbcTemplate.queryForObject(sql,
namedParameters, Integer.class) ;
```

MapSqlParameterSource 是一个类似 Map 风格的类，包含 Key-Value，Key 就是 SQL 中的参数名字，

在 SQL 语句中，不再使用“?”，而是使用了“:”开头的参数名字，如“:deptId”。

Spring 提供了 SqlParameterSource 类来封装任意的 JavaBean，为 NamedParameterJdbcTemplate 提供参数，JavaBean 的属性名将作为 SQL 的参数名字：

```
User user = new User();
user.setId(1);
user.setDepartmentId(1);
String sql = "update user set name=:name and departmet_id=:departmentId where
id = :id";
SqlParameterSource source = new BeanPropertySqlParameterSource(user);
namedParameterJdbcTemplate.update(sql, source);
```

## 5.3 BeetlSQL 介绍

上一节介绍了 Spring JDBC Template，封装了 JDBC 数据库访问代码，提高了开发效率和可维护性，然而对于企业应用来说，JDBC Template 还有很多未解决的问题，比如：

- 实体的增、删、改，以及简单的查询都需要编写代码，大多数 DAO 框架中的这种操作已经具有不需编写任何代码的能力。
- SQL 没有更好地管理起来，Java 代码中拼写 SQL 语句，一旦 SQL 调整，会调整 Java 代码，在有些应用系统中，拼写 SQL 的代码能有 50 行到 500 行，甚至上千行都有。
- 在 Java 中编写 SQL，对于不同的数据库、不同 SQL，只能硬编码，根据数据库类型来调整 SQL。
- JDBC Template 缺少 ORM 查询功能。

BeetlSQL 是笔者开发的一款功能齐全 DAO 工具，集中了下一章要提到的 JPA 那种内置多种 DAO 操作功能的优点，也吸取了 MyBatis 这样 SQL 集中管理的优点。事实上，越来越多的 DAO 工具，以及 JPA、MyBatis 本身也在向这两个方向不断完善。BeetlSQL 就是一个很好的工具，上手快、功能强大，既能应付简单的 SQL 操作，也能应付上百行 SQL 的复杂操作，可满足企业的各种需求。

## 5.3.1 BeetlSQL 功能概览

BeetlSQL 是一个全功能 DAO 工具，同时具有 Hibernate 和 MyBatis 的优点，适用于承认以 SQL 为中心，同时又需要工具能自动生成大量常用的 SQL 的应用。

- 开发效率
  - 无须注解，自动使用大量内置 SQL，轻易完成增删改查功能，节省约 50%的开发工作量；
  - 数据模型支持 POJO，也支持 Map/List 这种快速模型，还支持混合模型；
  - SQL 模板基于 Beetl 实现，更容易编写和调试，以及进行扩展；
  - 可以针对单个表（或者视图）代码生成 POJO 类和 SQL 模板，甚至是整个数据库，能减少代码编写工作量；
  - 支持 ORM 查询功能。
- 维护性
  - SQL 写在 Markdown 文件中，同时方便程序开发和数据库 SQL 调试；
  - 可以自动将 SQL 文件映射为 DAO 接口类；
  - 具备 Interceptor 功能，可以调试、性能诊断 SQL，以及扩展其他功能。
- 灵活直观地支持一对一、一对多、多对多关系映射而不引入复杂的 ORMapping 概念和技术。
- 其他
  - 内置支持主从数据库的开源工具；
  - 支持跨数据库平台，将开发者所要完成的工作减到最小，目前跨数据库支持 MySQL、PostgreSQL、Oracle、SQL Server、H2、SQLite、DB2。

## 5.3.2 添加 Maven 依赖

BeetlSQL 提供 starter 来快速集成 Spring Boot，在 pom 中需要添加如下依赖：

```
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl-framework-starter</artifactId>
  <version>1.1.15.RELEASE</version>
</dependency>
```

beetl-framework-starter 会自动集成 Spring Boot 已经配置好的数据源，可以参考 5.1 节来配置数据源。

### 5.3.3　配置 BeetlSQL

beetl-framework-starter 会读取 application.properties 如下配置：

- beetlsql.sqlPath，默认为/sql，作为存放 SQL 文件的根目录，位于/resources/sql 目录下。
- beetlsql.nameConversion，默认是 org.beetl.sql.core.UnderlinedNameConversion，能将下画线分割的数据库命名风格转化为 Java 驼峰命名风格，还有常用的 DefaultNameConversion，数据库命名完全和 Java 命名一致，以及 JPA2NameConversion，兼容 JPA 命名。
- beetl-beetlsql.dev，默认是 true，即向控制台输出执行时的 SQL，包括参数、执行时间及执行的位置，每次修改 SQL 文件时，自动检测 SQL 文件修改。
- beetlsql.daoSuffix，默认为 Dao。5.3.6 节会介绍 Mapper 功能，任何接口继承了 BaseMapper 接口，将自动具备实体内置的 CRUD 功能。daoSuffix 选项会在 Spring Boot 启动的时候，自动扫描以 Dao 结尾的接口，自动注册为 Spring 管理的 Dao 类，你可以在任何地方自动注入这个 Dao 类，比如在 Service 类中。
- beetlsql.basePackage，默认为 com，此选项配置 beetlsql.daoSuffix 来自动扫描 com 包及其子包下的所有以 Dao 结尾的 Mapper 类。以本章例子而言，你可以配置 “com.bee.sample.ch5.dao”。
- beetlsql.dbStyle，数据库风格，默认是 org.beetl.sql.core.db.MySqlStyle，对应不同的数据库，其他还有 OracleStyle、PostgresStyle、SqlServerStyle、DB2SqlStyle、SQLiteStyle、H2Style。

通常需要配置的是 beetlsql.nameConversion，根据数据库命名规则选择 NameConversion，还有根据数据库选择 Style 类，如果这两项配置不正确，会导致 BeetlSQL 运行出错。配置这两项完毕，可以在 Spring 管理的 Bean 中注入 SqlManager 来完成数据库访问操作，在 Service 中使用 SqlManager：

```
@Service
public UserServiceImpl implements UserService {
  @Autowried
  SqlManager sqlManager ;
}
```

### 5.3.4 SQLManager

SQLManager 是 BeetlSQL 的核心类，提供了所有的数据访问操作，比如根据主键查找实体操作，可调用 unique 方法：

```
Integer pk = 1;
User user = sqlManager.unique(User.class, pk);
```

也可以调用 single 方法，如果未查找到，仅仅返回 null：

```
User user = sqlManager.single(User.class, pk);
```

添加一个实体，调用 insert 方法：

```
User user = new User();
user.setDepartmentId(1);
user.setCreateTime(new Date());
user.setName("xiandafu");
sqlManager.insert(user);
```

USER 表的 ID 是自增的，如果想获取到自增主键，可以传入 true 参数：

```
sqlManager.insert(user,true);
```

根据主键更新实体：

```
User user = new User();
user.setId(1);
......
user.setName("NewName");
sqlManager.updateById(user);
```

只更新有值的属性：

```
User user = new User();
user.setId(1);
user.setName("NewName");
sqlManager.updateTemplateById(user);
```

上面的操作都会在控制台中打印类执行日志，以 updateTemplateById 方法调用为例，有如下输出：

```
┌──────── Debug [user._gen_updateTemplateById] ───
├ SQL:    update `user` set `name`=? where `id` = ?
├ 参数:     [NewName, 1]
├ 位置:
com.bee.sample.ch5.BeetlSqlTest.updateTemplateUser(BeetlSqlTest.java:74)
├ 时间:     48ms
├ 更新:     [1]
└──────── Debug [user._gen_updateTemplateById] ───
```

- 第一行有 SQL 标示，是 user._gen_updateTemplateById，BeetlSQL 是以 SQL 为中心的 DAO 工具，每个 SQL 都有对应 SQL 的唯一标示。对于 BeetlSQL 内置的 SQL 来说，也有唯一标示，通常以“对象名._gen_”开头。
- 第二行包含了 SQL 语句，对于过长的 SQL，比如使用 Markdown 保存的 SQL 经常多到数百行，会截取一部分显示。
- 第三行包含了传入的 Java 参数，通过调用参数的 toString 方法显示出来，日期类型统一用“yyyy-MM-dd HH:mm:ss.SSS”格式化输出。
- 第四行包含了 BeetlSQL 执行的位置，可以方便、快速地定位问题。在 IDE 环境下，直接点击这一行，能跳转到代码执行处。
- 第五行是 SQL 在数据库中执行的时间。
- 第六行包含执行结果、更新语句返回更新影响的行数，如果是查询语句，返回集合则打印出集合长度，如果返回单个对象，则打印对象。

> 这里 User 对象只是一个普通类，类名和属性名通过 NameConversion 与数据库表名和列名一一对应。

## 5.3.5 使用 SQL 文件

通常一个项目中还是有不少复杂的 SQL（有的只有五六行，有的甚至有数百行），放在单独的 SQL 文件中更容易编写和维护，需要在/resources/sql 目录下创建 user.md 文件，内容如下：

```
selectSample
```

```
===
* 一个简单的查询例子
* 根据用户名查询用户
  select * from user where 1=1
  @if(!isEmpty(name)){
  and name = #name#
  @}
```

这个 SQL 语句包含了一个简单的查询，并根据 name 变量是否有值来确定 SQL 语句。

以下是编写 SQL 模板的一些简单说明：

- 采用 md 格式，“===”上面是 SQL 语句在文件中的唯一标示，下面则是 SQL 语句；
- “===”下面可以放多行注释，采用“*”开头；
- @和回车符号是定界符号，可以在里面写 Beetl 语句，如条件判断语句；
- “#”是占位符，生成 SQL 语句的时候，占位符将输出“?”，并记录此位置的值，如果想直接输出值，需要用 text 函数，或者任何以 db 开头的函数，引擎则认为是直接输出文本；
- isEmpty 是 Beetl 的一个函数，用来判断变量是否为空或者是否不存在；
- 文件名约定为类名，首字母小写。

关于 Beetl 语法，请参考第 4 章。

selectSample 是 SQL 语句的标识，可以联合文件名来引用 SQL，比如 user.selectSample，代码如下：

```
User query = new User();
query.setName("NewName");
List<User> list = sqlManager.select("user.selectSample", User.class,query);
```

sqlManager 会查询/sql 目录下的 user.md（或者 SQL 后缀）文件，找到 selectSample 对应的 SQL 模板语句，User.class 参数表示要将数据库的查询结果集映射到 User 类上。也可以指定任何类，SQLManager 将结果集映射到类上，会选择结果集与类的共同属性进行映射。

参数 query 包含了 name 属性，且赋值为“NewName”，因此，最后的查询结果如下：

```
┏━━━━━━━━━━ Debug [user.selectSample] ━━━━━━
┣ SQL:     select * from user where 1=1 and name = ?
┣ 参数:     [NewName]
```

```
├ 位置:    com.bee.sample.ch5.BeetlSqlTest.queryUser(BeetlSqlTest.java:81)
├ 时间:    22ms
├ 结果:    [1]
└───────── Debug [user.selectSample] ─────
```

可以向 SQL 模板传入一个 Map 而不是 POJO，查询代码是这样的：

```
public static void queryUserByMap(SQLManager sqlManager){
  Map paras = new HashMap();
  paras.put("name", "NewName");
  List<User> list = sqlManager.select("user.selectSample", User.class,paras);
}
```

## 5.3.6 Mapper

对于 md 文件中的 SQL 查询，使用 sqlId 不方便维护和使用，BeetlSQL 提供 Mapper 功能，能将 md 文件的 sqlId 映射为方法名。可以创建一个接口，继承 BaseMapper 接口，并且添加一个“selectSample”方法：

```
package com.bee.sample.ch5.dao;
import java.util.List;
import org.beetl.sql.core.mapper.BaseMapper;
import com.bee.sample.ch5.entity.User;

public interface UserDao extends BaseMapper<User> {
   public List<User> selectSample(User query);
}
```

UserDao 可以在 Spring Boot 启动的时候自动注入，因此可以这么用：

```
@Service
public class UserServiceImpl implements UserService {
  @Autowired UserDao userDao;

  public List<User> getUserByName(String name) {
    User query = new User();
    query.setName("NewName");
```

```
    List<User> list = userDao.selectSample(query);
  }
}
```

selectSample 语句实际上需要一个 name 参数，如果传入 User 对象，尽管 SQLManager 可以读取其属性 name，但接口并不方便阅读，因此，mapper 也可以改成如下形式：

```
public interface UserDao extends BaseMapper<User> {
  public List<User> selectSample(String name);
}
```

BaseMapper 是 BeetlSQL 提供的一个内置 Dao 接口，内置了多种增删改查方法，包含如下：

- Insert，插入实体对象到数据库；
- insertTemplate，插入实体到数据库，只插入非 null 属性；
- insertBatch，批量插入实体；
- updateById，按照主键更新实体；
- updateTemplateById，按照主键更新实体，只更新非 null 的属性；
- deleteById，按照主键删除实体，一些大型应用严格来说禁止删除，参考 BeetlSQL 官网了解如何定制 BaseMapper，不提供 deleteById 方法；
- unique，根据主键查询实体，如果没有找到，抛出 Runtime 异常；
- single，根据主键查询实体，如果没有找到，返回 null；
- lock， 根据主键使用数据库悲观锁，相当于 select * from table where id = ? for update；
- all，返回所有实体；
- allCount，所有实体的个数；
- template，查询符合模板的所有实体；
- templateOne，查询符合模板的第一个实体；
- execute，执行一个 JDBC 查询，在 Java 中提供一个 SQL 语句；
- executeUpdate，执行一个 JDBC 更新操作，在 Java 中提供一个 SQL 语句。

这些内置方法都会在 5.4 节详细讲述，另外，如果你想自定义 Spring Boot 应用的 BaseMapper，可以参考 BeetlSQL 官网定制自己风格的 BaseMapper 类。

### 5.3.7 使用实体

BeetlSQL 对实体没有特别的要求，只要能按照 NameConverson 指定的命名约定对应一个普通的 JavaBean 即可。对于 User 表，对应了如下 JavaBean：

```
// 通过 SQLManager.gen 方法生成的类
public class User  implements Serializable{
   private Integer id ;
   private Integer departmentId ;
   // 名称
   private String name ;
   // 创建时间
   private Date createTime ;
   // 忽略 getter 和 setter 方法
}
```

BeetlSQL 并不要求列名能和实体的属性名一一对应，BeetlSQL 在 SQL 操作的时候，只会选取列名和属性名的“交集”来操作。

除了实体，还有大量查询返回额外属性，可以考虑新增 Java 对象来映射 SQL 查询结果，比如某个查询返回用户一年的薪水，假设返回结果的列名是 YEAR_INCOME，有三种办法来解决：

- 在 User 对象上新增一个属性 yearIncome；
- 新建一个对象 UserIncome，包含了用户属性和 yearIncome；
- User 对象继承 TailBean，TailBean 有一个类似 Map 结构可以设置查询返回的额外属性到 Map 中，后面讲到的 ORM 查询返回的其他关联实体也可以放到 Map 中，将在 5.5.6 节中详细介绍。

## 5.4 SQLManager 内置 CRUD

SQLManager 内置 API 是指不需要自己写 SQL 就能完成对象的增删改查，通常能完成项目 50%的数据库访问需求，根据传入的 POJO 对象来生成内置的 CRUD 语句。

### 5.4.1 内置的插入 API

- public void insert(Object paras)，插入 paras 到 paras 关联的表；

- public void insert(Object paras,boolean autoAssignKey)，插入 paras 到 paras 对象关联的表，并且指定是否自动将数据库主键赋值到 parask 中，此 API 适用于带有自增主键的表；
- public void insertTemplate(Object paras)，插入 paras 到 paras 关联的表，忽略为 null 值或者为空值的属性；
- public void insertTemplate(Object paras,boolean autoAssignKey)，插入 paras 到 paras 对象关联的表，并且指定是否自动将数据库主键赋值到 paras 中，忽略为 null 值或者为空值的属性，调用此方法，对应的数据库必须主键自增；
- public void insert(Class<?> clazz,Object paras)，插入 paras 到 clazz 关联的表；
- public int insert(Class clazz,Object paras,boolean autoAssignKey) ，插入 paras 到 clazz 关联的表，并且指定是否自动将数据库主键赋值到 paras 中，调用此方法，对应的数据库必须主键自增；
- public void insertBatch(Class clazz,List<?> list)，批量插入数据。

## 5.4.2 内置的更新（删除）API

- public int updateById(Object obj)，根据主键更新，所有值参与更新；
- public int updateTemplateById(Object obj)，根据主键更新，属性为 null 的不会更新；
- public int updateBatchTemplateById(Class clazz,List<?> list)，批量根据主键更新，属性为 null 的不会更新；
- public int updateTemplateById(Class<?> clazz,Map paras)，根据主键更新，主键通过 clazz 的 annotation 表示如果没有，则认为属性 id 是主键，属性为 null 的不会更新；
- public int[]updateByIdBatch(List<?> list)，批量更新。

## 5.4.3 内置的查询 API

- public List all(Class clazz)，查询出所有结果集；
- public List all(Class clazz, int start, int size)，查询指定范围的结果集；
- public int allCount(Class<?> clazz)，总数查询；
- public List template(T t)，根据模板查询，返回所有符合这个模板的数据；
- public T templateOne(T t)，根据模板查询，返回一条结果，如果没有找到，返回 null；

- public List template(T t,int start,int size)，同上，可以指定范围；
- public T unique(Class clazz,Object pk)，根据主键查询，如果未找到，抛出异常；
- public T single(Class clazz,Object pk)，根据主键查询，如果未找到，返回 null；
- public long templateCount(T t)，获取符合条件的个数。

> **注意：**默认的翻页是从 1 开始的，BeetlSQL 将会根据数据库来翻译成数据库翻页的 offset，比如，MySQL 数据库将转化成 0，而 Oracle 数据库则保持不变，这有利于跨数据库。
>
> BeetlSQL 提供 PageQuery 来处理翻页查询，详情参考 5.5.4 节。

### 5.4.4 代码生成方法

如果已经创建好数据表，BeetlSQL 能自动生成表对应的实体、Dao 操作和 md 文件，md 文件里包含了相关 SQL 查询语句。

- genPojoCodeToConsole(String table)，根据表名生成 POJO 类，输出到控制台。
- genSQLTemplateToConsole(String table)，生成查询条件，更新 SQL 等语句，输出到控制台。
- genPojoCode(String table,String pkg,GenConfig config)，生成 POJO 代码到项目工程中，pkg 指定了包名，GenConfig 指定了生成的细节，比如：
  - baseClass，POJO 的基类，默认为 java.lang.Object；
  - spaceCount，格式控制，默认为 4 个空格；
  - preferBigDecimal，默认为 true，对于浮点类型，对应的 Java 类型使用 BigDecimal，否则使用 Double；
  - preferDate，日期类型，默认使用 java.util.Date，否则使用 Timestamp。

> 使用 Double 类型实在是一个糟糕的办法，如果 POJO 使用 Double，则从数据库结果集取出来，映射到 Double 类型，有可能会丢失一些精度，因此建议使用 BigDecimal。

- genSQLFile(String table)，生成指定表对应的 md 文件。
- genAll(String pkg,GenConfig config,GenFilter filter)，生成所有的 POJO 代码和 SQL 模板，GenFilter 用来过滤，仅生成 user 表对应的 POJO、Dao 和 md 文件，代码如下：

```
sqlManager.genAll("com.test", new GenConfig(), new GenFilter(){
```

```
  public boolean accept(String tableName){
    if(tableName.equalsIgnoreCase("user")){
      return true;
    }else{
      return false;
    }
  }
});
```

谨慎使用 genAll，因为会覆盖已经生成好的 Dao 和 POJO，以及 md 文件。

## 5.5 使用 sqlId

有些业务系统的数据库操作比较复杂，通常 SQL 有很多行，拼接 SQL 的条件也较为复杂，像 Spring Data JPA 那样用简单命名约定来实现的 SQL 查询很难应用在互联网、电信、金融、ERP 等行业，因此，把原生 SQL 写到专门文件中能提高开发效率和便于维护。BeetlSQL 支持将 SQL 放到 md 中，有以下好处：

- md 格式本身比较简单，适合阅读书写；
- 与 BeetlSQL 类似的 MyBatis 采用 XML 格式来维护，md 比 XML 更合适 SQL 语句，因为没有像 XML 那样过多的格式控制，还有 SQL 中的“<”符号是特殊符号，放在 XML 中必须转义；
- md 编辑器比较丰富，也支持导出到其他格式，方便项目所有人员阅读。

### 5.5.1 md 文件命名

针对数据库中每一个表或者视图对应的实体 POJO，都可以创建一个同名且首字母小写的文件名，放在默认的 SQL 目录下。比如数据库 SYS_USER，对应的 POJO 是 SysUser，可以在 SQL 目录下创建 sysUser.md 文件。

这样，sqlId 是 sysUser.select，将访问 sysUser.md 文件中的 SQL。

SQL 文件以 md 为扩展名，也支持以 sql 为扩展名。

> 以上通过 sqlId 来找到对应文件是 BeetlSQL 的默认规则，可以实现 SQLIdNameConversion 接口来自己定义一个匹配规则。

实际上，BeetlSQL 并非直接寻找 SQL 目录下的 md 文件，它会根据数据库类型首先寻找

SQL 目录下是否存在与数据库同名的文件夹，如果有，则先在其目录下寻找匹配的 sqlId，如果没有找到，才会在 SQL 目录下寻找。比如你的应用同时支持 MySQL 和 Oracle，刚好关于用户的某个统计需要用到两种数据库的不同分析函数，因此，你可以在 MySQL 和 Oracle 目录下分别创建一个 user.md，然后将这个统计 SQL 写到各自的 md 文件中，BeetlSQL 运行时，会根据当前使用的数据库来优先寻找数据库下的 md 文件。

## 5.5.2　md 文件构成

md 文件由多个 SQL 片段构成，格式如下：

```
一些说明，可有可无

getXXX
===

* 注释，可有可无
* 最好为一些复杂的 SQL 添加注释

    select * from xxxx

updateXXX
===

* 注释

    update xxx  set ....
```

SQL 片段通过 md 的“===”分开，上面一行为 SQLId,下面为 SQL 的内容。也可以在“===”下面紧挨着放置一些注释，通过 md 的*号来表示这是注释。

SQL 语句可以任意长，如果你有一个很好的 md 编辑器，可以使用“'''sql”来标注 SQL 内容。在某些 md 编辑器中，会识别这是一个 SQL 片段，能高亮显示其中的 SQL 语法，当然，在 IDE 中，你的文件可以以 SQL 类型打开，同样能高亮显示 SQL 语法。

## 5.5.3　调用 sqlId

有了 sqlId，你可以通过 sqlManager 或者 Mapper 来调用：

- 查询类接口
  - public List select(String sqlId, Class clazz, Map/Object paras)，根据 sqlId 来查询，参数是 Map 或者 POJO。
  - public List select(String sqlId, Class clazz)，根据 sqlId 查询，无参数。
  - public List select(String sqlId, Class clazz，Map/Object paras, int start, int size)，指定范围查找，翻页查询参考下一节。
  - public T selectSingle(String id,Map/Object paras, Class target)，根据 sqlId 查询，输入是 Map 或者 POJO，将对应的唯一值映射成指定的 target 对象，如果未找到，则返回空。需要注意的是，有时候结果集本身可能为空，这时候建议使用 unique 查询。
  - public T selectUnique(String id,Map/Object paras, Class target)，根据 sqlId 查询，输入是 POJO 或者 Map，将对应的唯一值映射成指定的 target 对象，如果未找到，则抛出异常。
  - public Integer intValue(String id,Map/Object paras)，查询结果映射成 Integer，如果找不到，返回 null，输入是 Map 或者 POJO。类似的方法还有 longValue、bigDecimalValue 方法，内部实现是调用 selectSingle 方法。
- 插入方法
  - public int insert(String sqlId,Object paras,KeyHolder holder)，根据 sqlId 插入，并返回主键，主键 id 由 paras 对象指定，调用此方法，对应的数据库表必须主键自增。
  - public int insert(String sqlId,Object paras,KeyHolder holder,String keyName)，同上，主键由 keyName 指定，告诉 BeetlSQL 自增主键的列名，BeetlSQL 会按照 keyName 取一次自增主键。
  - public int insert(String sqlId,Map paras,KeyHolder holder,String keyName)，同上，参数通过 Map 提供。
- 更新方法
  - public int update(String sqlId, Map/Object obj)，根据 sqlId 更新。
  - public int[]updateBatch(String sqlId,List<?> list)，批量更新，返回更新结果。
  - public int[]updateBatch(String sqlId,Map[]maps)，批量更新，参数是数组，元素类型是 Map。

## 5.5.4 翻页查询

```
public void pageQuery(String sqlId,Class clazz,PageQuery query)
```

BeetlSQL 提供一个 PageQuery 对象用于 Spring Boot 应用的翻页查询。它为查询提供参数、查询范围、排序。BeetlSQL 假定有 sqlId 和 sqlId$count 两个 sqlId，并用这两个来翻页和查询结果总数，代码如下：

```
queryNewUser
===
select * from user order by id desc;

queryNewUser$count
===
select count(1) from user
```

大部分情况下都不需要两个 SQL 来完成，一个 SQL 也可以，要求使用 page 函数或者 pageTag 标签，这样才能同时获得查询结果集总数和当前查询的结果。

```
queryNewUser
===

 select
 @pageTag(){
 a.*,b.name role_name
 @}
 from user a left join b ...
```

以上 SQL 会在查询的时候转为两条 SQL 语句：

```
select count(1) from user a left join b...
select a.*,b.name role_name  from user a left join b...
```

如果字段较多，为了输出方便，也可以使用 pageTag；如果字段较少，用 page 函数也可以，具体参考 pageTag 和 page 函数说明。翻页代码如下：

```
// 从第一页开始查询，无参数
PageQuery query = new PageQuery();
```

```
query.setParas(xxx);
sql.pageQuery("user.queryNewUser", User.class,query);
// 查询结果
System.out.println(query.getTotalPage());
System.out.println(query.getTotalRow());
System.out.println(query.getPageNumber());
List<User> list = query.getList();
```

PageQuery 提供了以下方法：

- setParas(Object paras)，设置查询用的参数，可以是 Map 或者 POJO。
- setPara(Sting key, Object Value)，设置查询用的参数。
- setPageSize，设置每页大小，默认是 20 条。
- setPageNumber，设置查询页数。
- setOrderBy，设置查询结果排序方式，如“id desc”。

**跨数据库支持**

如果打算使用 PageQuery 做翻页，且只想提供一个 SQL 语句+page 函数，如果考虑到跨数据库，应该不要在这个 SQL 语句里包含排序，因为大部分数据库都不支持。page 函数生成的查询总数 SQL 语句因为包含了 order by，在大部分数据库中都是会报错的，比如 select count(1) from user order by name，在 SQL Server、MySQL、PostgreSQL 中都会出错，Oracle 允许这种情况。因此，如果你要使用一条 SQL 语句+page 函数，建议排序用 PageQuery，PageQuery 对象中有排序属性 order by，可用于排序，而不是放在 SQL 语句中。

BeetlSQL 也提供了标签函数 pageIgnoreTag，可以用在翻页查询中，当查询用于统计总数的时候，会忽略标签体内容，如：

```
select page("*") from xxx
@pageIgnoreTag{
order by id
@}
```

以上语句在求总数的时候，会翻译成 select count(1) from xxx。

## 5.5.5 TailBean

对于 SQL 查询，可以将结果集映射到 POJO 上，可以为复杂的结果集专门写一个 POJO，

也可以在实体对象上添加额外的属性来保存映射。BeetlSQL 也提供查询结果映射到 Map 上。

BeetlSQL 提供 TailBean 类，POJO 类继承此类后，SQL 查询结果集映射不到的字段将会放到此类中，称之为混合模型，可以通过 get(xxx)来获取其值：

```
public class User extends TailBean{

}
```

如果你有一个 SQL 查询如下所示，查询结果集多出了一个 department_name：

```
select u.*,d.name department_name from user u left join department d....
```

BeetlSQL 在将结果集映射到 User 对象的时候，department_name 没有对应的属性 departmentName（假设用的是 UnderlinedNameConversion），则 BeetlSQL 会将其值放到 TailBean 里，因此可以通过 get 方法获取：

```
List<User> list = sqlManager.select("user.selectSample", User.class,paras);
for(User user:list){
  System.out.println(user.get("departmentName"));
}
```

通过 get 方法有时候不利于代码阅读和维护，可以在 user 对象中创建一个 getDepartmentName 方法：

```
public String getDepartmentName(){
  return (String)user.get("departmentName");
}
```

## 5.5.6 ORM 查询

BeetlSQL 关系映射是在 SQL 语句中通过 orm.single、orm.many、orm.lazySingle、orm.lazyMany 函数进行声明的。BeetlSQL 会根据这些声明来完成关系映射，它与 JPA 声明实体关系来实现 ORM 查询不一样，BeetlSQL 完成的 ORM 查询只是在主查询完毕后，发出的一次或者多次查询，理解起来比传统的 ORM 工具容易。

orm.single、orm.many、orm.lazySingle、orm.lazyMany 函数名字本身说明了是一对一还是一对多或者多对多，以及是直接加载还是懒加载。函数可以放在 SQL 语句的任何地方，建议放到

头部或者尾部，参数格式有两种形式：

- 使用模板方式查询关系对象。orm.single({"departmentId","id"},"Department")的第一个参数声明了关系映射，即 SQL 查询结果的属性（非数据库字段名），对应关系表的查询属性，如 User 对象中，departmentId 对应 Department 对象的 id，BeetlSQL 会根据此关系发起一次 template 查询。映射的结果集放在第二个参数 Department 类中，如果 Department 与 User 类在同一个包下，可以省略包名，否则需要加上类包名。
- 使用 sqlId 来查询关系对象，orm.single({"departmentId","id"},"user.selectDepatment","Department")的第一个参数还是映射关系，第二个参数是 SQL 查询 id，BeetlSQL 将查询此 SQL 语句，将结果集放到第三个参数 Department 类。

lazy 意味着在调用的时候再加载。如果在事务外调用，并不会像 Hibernate、JPA 那样报错，BeetlSQL 会再用一个数据库连接去查询。一般来说，如果业务代码确定要用，建议不使用 lazy 方式。因为 lazy 不会查询优化，性能可能差一些。

以上查询关系对象均放到 TailBean 中，名称以小写类名开头，如：

```
    User user =
sqlManager.select("user.selectUserAndDepartment",User.class,paras);
    Department dept = user.get("department");
```

以下是一个例子，假设 user 表与 department 表示一对一关系，user.departmentId 对应于 department.id，因此关系映射是{"departmentId":"id"}，user 与 role 表示多对多关系，通过 user_role 进行关联。

```
selectUserAndDepartment
===
    select * from user where user_id=#userId#
    @ orm.single({"departmentId":"id"},"Department");
    @ orm.many({"id":"userId"},"user.selectRole","Role");

user.selectRole
===

    select r.* from user_role ur left join role r on ur.role_id=r.id
    where ur.user_id=#userId#
```

Java 代码如下：

```
User user =
sqlManager.select("user.selectUserAndDepartment",User.class,paras);
Department dept = user.get("department");
List<Role> roles = user.get("role");
```

> BeetlSQL 也支持通过在 POJO 上联合使用注解@OrmQuery 和 OrmCondition 来完成 ORM 查询，具体参考 BeetlSQL 官网文档。

## 5.5.7 其他 API

- 直接执行 SQL 模板语句
  - public List execute(String sqlTemplate,Class clazz, Object/Map paras);
  - public int executeUpdate(String sqlTemplate,Object/Map parasparas)，返回成功执行条数。

  sqlTemplate 是 SQL 模板：

  ```
  List list = sqlManager.execute("select * from user where
  name=#name#",User.class,para);
  ```

- 也可以使用 SQLReady 直接执行已经构建好的 JDBC SQL，SQLReady 对象包含了 SQL 和参数：
  - public List execute(SQLReady p,Class clazz)，查询接口；
  - public int executeUpdate(SQLReady p)，更新接口；
  - public PageQuery execute(SQLReady p, Class clazz, PageQuery pageQuery)，执行 JDBC SQL，实现翻页查询，翻页参数在 PageQuery 中，SQL 参数在 SQLReady 中。

SQLReady 包含了 JDBC SQL 和相应的参数，如：

```
SQLReady ready =new SQLReady("update user set age=?",age);
sqlManager.executeUpdate(ready);
// 从第一页开始查找，每页 10 个
PageQuery query = new PagaeQuery(1,10);
PageQuery page = sqlManager.execute(new SQLReady("select * from
user"),User.class,)
```

### 5.5.8 Mapper 详解

SQLManager 提供了所有需要知道的 API，但通过 sqlId 访问 SQL 有时候还是很麻烦，因为需要手动输入字符串，另外参数不是 Map 就是 Object，对代码理解没有好处。BeetlSQL 支持 Mapper，将 SQL 文件映射到一个 interface 中。示例如下：

```
public interface UserDao extends BaseMapper<User> {

    // 使用"user.getCount"语句，无参数
    public int getCount();
    // 使用"user.setUserStatus"语句
    public void setUserStatus(Map paras); // 更新用户状态
    public void setUserAnotherStatus(User user); // 更新用户状态
    // 使用"user.findById"，传入参数 id
    public User findById(Integer id);
    public User findByIdAndStatus( Integer id,Integer status);
    // 翻页查询，使用"user.queryNewUser"，查询结果在 query 对象中
    public void queryNewUser(PageQuery query) ;
    // 使用 sqlready
    @Sql(value=" update user set age = ? where id = ? ")
    public void updateAge(int age,int id);
    // 使用 sqlready 查询，返回字符串
    @Sql(value=" select name from user")
    public List<String> allNames();

}
```

- Interface 继承 BaseMapper，这样可以使用 BaseMapper 的一些公共方法，如 insert、unique、single、updateById、deleteById 等。
- Interface 中的方法名与 SQL 文件中的 SQL 片段对应，如果方法名对应错了，会在调用的时候报错找不到 SQL。
- 方法参数可以是一个 Object，或者是 Map，这样，BeetlSQL 自动识别为 SQL 的参数，也可以使用注解@Param 来标注，或者混合这两种情况，如：

```
public void setUserStatus(Map paras,@Param("name") String name);
}
```

注意 BeetlSQL 会根据方法对应的 SQL 语句解析开头，如果是 select 开头，就认为是 select 操作，同理还有 update、delete、insert。如果 SQL 模板不是以这些关键字开头，则需要使用注解@SqlStatement。

```
@SqlStatement(type=SqlStatementType.INSERT)
// 添加用户，如果 newUser 对应的 SQL 语句不是 insert 开头
public KeyHolder newUser(User user);
```

对于 Mapper 涉及的查询来说，会将查询结果映射到返回值上，有以下规则：

- Mapper 方法的返回值应该与查询结果对应，如果查询结果是日期，方法返回值也应该是日期（Date 或者 Timestamp）。

```
// 不需要注解说明返回类型
public Date getMaxDate(User query);
```

- 如果返回值是 List 或实体本身集合，不需要泛型，如果是其他类型，则使用泛型加以说明。

```
public List<Integer> getTop10Id();
```

- 翻页查询 PageQuery 不需要做类型说明，默认返回就是实体本身，如果返回的是其他类型，应加上泛型。

```
// 翻页查询，默认返回实体对象
public void queryUser(PageQuery query);
public void queryUserName(PageQuery<String> query);
```

## 5.6 BeetlSQL 的其他功能

BeetlSQL 还有一些高级功能，本书不会全部讲解，本节介绍常用的高级知识，需要深入了解可以通过 ibeetl.com 官网深入了解 BeetlSQL，本节主要介绍如下内容。

- SQL 模板语句常用函数和标签；
- 主键；
- BeetlSQL 常用注解；
- NameConvesion，命名转化类；

- 悲观锁和乐观锁。

### 5.6.1 常用函数和标签

可以在 SQL 模板中使用以下函数：

- isEmpty，判断变量是否存在，以及是否为 null；
- print，println 输出变量；
- debug，将变量值输出到控制台；
- text，占位符中输出变量对应的文本；
- join，用逗号连接数组或者集合，通常用于 in 操作；

```
select * from user where status in ( #join(ids)#)
-- 输出成  select * from user where status in (?,?,?)
```

- use，可以重用当前 SQL 文件的一段模板；

```
condition
===
where 1=1 and name = #name#


selectUser
===
select * from user #use("condition")#
```

use 参数也允许传入参数到 SQL 片段，比如：

```
select * from user #use("condition",{'para1':para})#
```

globalUse 用法同 use，参数是其他文件的 SQL 片段：

```
select * from user  where #globalUse("share.accessControl")#
```

将访问 share.md 的 accessControl SQL 片段。

- trim 标签，类似 MyBatis 的 trim 语法，能删除标签内的前后缀，默认不传参数，删除的是标签体最后的逗号。

```
updateStatus
===
update user set
@trim(){
    @if(!isEmpty(age){
age = #age# ,
    @} if(!isEmpty(status){
status = #status#,
    @}
@}
where id = #id#
```

上述语句会删除最后的逗号，可以使用：

```
@trim({"suffix":","}){

@}
```

- pageTag 用于对 SQL 进行翻页，使得在求总数的时候输出成 count(1)：

```
queryNewUser
===
select
@pageTag(){
id,name,status
@}
from user
```

也可以使用 page 函数：

```
queryNewUser
===
select #page("*")# from user
```

## 5.6.2 主键设置

BeetlSQL 支持三种主键：

- 自增主键，@AutoID，作用于主键字段或者 getter 方法上，表示对应数据库的自增主键列。如果你用的是 MySQL，且字段名恰好是 id，则可以省略此主键。
- 序列主键，@SeqID (name="seqname")，作用于主键字段或者 getter 方法上，对应于数据库序列名字。
- 程序赋值主键，@AssignID 表示主键是程序赋值，@AssignID 也支持某种算法交给 BeetlSQL 自动赋值，比如 BeetlSQL 内置了一个 Snowflake 算法（一个分布式的 id 算法，比 UUID 占用空间小，且 id 具有一定业务含义）。

```
@AssignID("simple")
public long getId(){}
```

simple 对应于 BeetlSQL 内置的 Snowflake 算法，当调用 BeetlSQL 的 insert 方法后，BeetlSQL 将自动按照 Snowflake 赋值，你可以实现自己的算法并添加到 SQLManager 中，比如：

```
sqlManager.addIdAutonGen("yourName",new IDAutoGen(){
  public T nextID(String params){
    return .....
  }
})
```

@AutoID 和@SeqID 可以叠加使用，不同的数据库 Style 只关心自己支持的注解，比如 MySqlStyle 关注@AutoID，OracleStyle 使用@SeqId。

> 为了更好地支持跨数据库操作，建议使用@AssignId("xxx")，xxx 是自定义的主键。

BeetlSQL 支持复合主键，这样主键就是其对象本身，比如 User 包含 id1、id2 两个主键，按照主键查询，代码如下：

```
User key = new User();
key.setId1();
key.setId2();
User user = sqlManager.unique(key);
```

### 5.6.3 BeetlSQL 注解

除了主键涉及的注解@AssignID、@AutoID 和@SeqID，BeetlSQL 还支持一些常用注解：

- @InsertLgnore，内置插入的时候忽略此属性。
- @UpdateLgnore，内置更新的时候忽略此属性。
- @EnumMapping，如果属性是枚举，可以通过此注解来指定如何将数据库的值转化为枚举。

```
@EnumMapping("value")
public enum Color {
 RED("RED",1),BLUE ("BLUE",2);
 private String name;
 private int value;
 private Color(String name, int value) {
  this.name = name;
  this.value = value;
 }
 // 省略 getter setter 方法

}
```

@EnumMapping 中的属性“value”指明了转化属性，BeetlSQL 会根据数据返回的值来匹配对应的枚举值，同样会将枚举的该属性的值保存到数据库中。

- @Table，指明数据库表名和 Java 类名存在对应关系，通常用于 BeetlSQL 自带的 NameConversion 不能满足需求的情况下，比如数据库表名是 T_SYS_USER，想对应到 SysUser 类，可以使用@Table。

```
@Table(name="T_SYS_USER")
public class QueryUser{

}
```

- @SqlResource，作用在 Mapper 接口上，说明 md 文件的位置，默认以 Spring Boot 应用的 resources/sql 作为根目录，可以通过此注解指定根目录下的某一个位置。

```
@SqlResource("platform.sysDict")
public interface SysDictDao extends BaseMapper<SysDict> {
  public List<SysDict> findAllList(@Param(value = "type") String type);
}
```

以上 findAllList 方法对应的 SQL 位于 resources/sql/platform/sysDict.md(sql)中。

### 5.6.4 NameConversion

BeetlSQL 支持以下命名转化类：

- UnderlinedNameConversion，将数据库表和列名字全部小写，删除下画线，采用 Java 驼峰命名，比如 SYS_USER，转为 SysUser，反之亦然。
- DefaultNameConversion，数据库表和列名字不做任何变化，比如数据库叫 SysUser，列名字是 userId，则刚好能对应上 Java 的 SysUser 类，以及 userId 属性。
- JPA2NameConversion，支持在属性上使用 JPA 注解。JPA2NameConversion 也可以传入一个 NameConversion 实现作为默认命名转化类，这样，没有使用 JPA 注解的属性可以直接使用默认命名转化类。

开发者也可以实现自己的 NameConversion 提供给 BeetlSQL 使用。

### 5.6.5 锁

BeetlSQL 支持悲观锁和乐观锁。对于悲观锁，有如下 API：

```
public <T> T lock(Class<T> clazz, Object pk)
```

或者 BaseMapper 中的 API：

```
T lock(Object key);
```

这两种方法都相当于向数据库发起了一个行级锁查询：

```
select * from table where id = ? for update
```

API 返回意味着获得了该悲观锁，如果没有获得悲观锁，则一直处于等待状态。
悲观锁的释放通常是在事务结束后，如果在其他框架下使用 BeetlSQL 的悲观锁，请确保是

在事务环境下调用的 lock 方法。

BeetlSQL 在实体属性上使用@Version 来提供乐观锁功能，此实体属性必须是 Integer 或者 Long：

```
public class Credit   implements Serializable{
    private Integer id ;
    private Integer balance ;
    @Version
    private Integer version ;
}
```

当调用内置的 updateById 或者 updateTemplateById 的时候，被@Version 注解的字段将作为 where 条件的一部分：

```
┏━━━━━━━ Debug [credit._gen_updateTemplateById] ━━━━
┣ SQL:   update `credit` set `balance`=?, `version`=`version`+1 where `id`
= ? and `version` = ?
┣ 参数:   [15, 1, 5]
┣ 位置:   org.beetl.sql.test.QuickTest.main(QuickTest.java:38)
┣ 时间:   4ms
┣ 更新:     [1]
┗━━━━━━━ Debug [credit._gen_updateTemplateById] ━━━━
```

如果更新失败，updateById 方法会返回 0，表示更新失败，说明在更新前，数据已经被其他应用更改，而且版本号增加了。如果方法返回 1，表示更新成功，同时版本号递增。

# 第 6 章 Spring Data JPA

访问数据库的方式有两个流派，一派以 SQL 为中心，在 JDBC 上做了一定程度的封装，比直接操作 JDBC 更加方便和便捷，我们在上一章介绍的 BeetlSQL 就属于这个流派，本书未介绍的流行 DAO 工具 MyBatis 也属于这个流派。

另外一个流派则是以 Java Entity 为中心，将实体和实体关系对应到数据库的表和表关系，这类工具通常就是 ORM（Object Relational Mapping）工具。对实体和实体关系的操作会映射到数据库操作。

本章将介绍 Spring Data JPA，它在 JPA 提供的简单语义上做了一定的封装，满足 CRUD 查询。同时，也会介绍 Spring Data，它为 Spring 框架对访问 SQL 和 NoSQL 数据库提供了一致的方式。考虑到 JPA 本身学习有一定的门槛，本章只重点学习 Spring Data JPA 的相关知识。

## 6.1 集成 Spring Data JPA

### 6.1.1 集成数据源

Spring Boot 集成 Spring Data JPA，需要在 pom 中添加 spring-boot-starter-data-jpa 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

本书采用 MySQL 作为例子，使用 HikariCP 作为数据源提供者，因此还需要在 pom 中添加依赖：

```
<dependency>
  <groupId>com.zaxxer</groupId>
  <artifactId>HikariCP</artifactId>
  <version>2.6.1</version>
</dependency>
<!-- 数据库驱动 -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>6.0.5</version>
</dependency>
```

为了在 Spring Boot 应用中使用 Spring Data JPA，需要通过 Java 来配置数据源。我们使用 Hikari 作为数据源：

```
@Configuration
public class DataSourceConfig {
  @Bean(name = "dataSource")
  public DataSource datasource(Environment env) {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl(env.getProperty("spring.datasource.url"));
    ds.setUsername(env.getProperty("spring.datasource.username"));
    ds.setPassword(env.getProperty("spring.datasource.password"));
    ds.setDriverClassName(env.getProperty("spring.datasource.driver-
class-name"));
    return ds;
  }
}
```

## 6.1.2 配置 JPA 支持

Spring JPA Data 采用了 Hibernate 实现，Hibernate 实现中有两个特性需要关注：

- spring.jpa.hibernate.ddl-auto，是否自动建库。默认为 none，Hibernate 能根据 Entity 类的定义自动生成表以及修改已有的表和表主外键设置等。我们在这里是先建好了数据库，采用默认值。对于内存数据，如 H2，Spring Boot 默认为 create-drop。
- spring.jpa.show-sql，是否自动打印 SQL，默认为 false，我们设置为 true，也就是 JPA 操作自动显示对应的数据库 SQL 语句。在刚学习 JPA 的时候，我们需要看到 JPA 操作对应的数据库 SQL。

在 application.properties 中可以配置这两个属性。

> 自动建表是个很酷的特性，但对于常规项目用处并不大，因为大多数项目都是在需求分析后建立物理模型。也就是先有数据库表设计，随后才是 Spring Boot 应用，因此数据库的 DDL 语句早已经具备。从另一方面讲，项目数据库管理人员、需求分析人员，甚至是项目经理并不喜欢 Hibernate 创建的表结构，比如，它的外键命名就很随机，不符合命名规范。自动建表对于小型项目或者工具类项目（如工作流引擎）还是很方便的。

## 6.1.3 创建 Entity

本节采用了上一章的数据库表作为例子，参考 5.1 节来了解 SQL 脚本。User 表对应的实体如下：

```
@Entity
public class User  {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer id ;
   // 部门
   @Column
   private String name ;
   // 创建时间
   @Column(name="create_time")
   private Date createTime ;

   @ManyToOne
   @JoinColumn(name="department_id")
   Department department;
```

```
    public User() {
      // JPA 要求实体必须有一个空的构造函数
    }
    // 忽略 getter 和 setter 方法
}
```

不同于 BeetlSQL 对 Entity 的“松散”要求，JPA 既然以 Entity 为中心，实体类就必须使用 @Entity 来注解，JPA 也提供了大量的注解来表明实体属性和关系。

- @Id，声明了一个属性将映射到数据库主键字段。主键生成策略由注解 @GeneratedValue 来指定。本例中，id 为自增主键，是一种简单的数据库主键生成策略，因此使用 GenerationType.IDENTITY 作为主键生成的策略。
- @Column，此注解表明属性对应到数据库的一个字段，且列名为 name 指定的名称。
- @ManyToOne，Many 指的是定义此属性的实体，即 User 实体，而 One 指的就是此注解所注解的属性，ManyToOne 说明对象 User 和 Department 的关系是多对一关系，多个用户属于一个部门。
- @JoinColumn，与 ManyToOne 搭配使用，说明外键字段是 department_id。

Department 对象的定义如下：

```
@Entity
public class Department{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id ;
    @Column
    private String name ;
    @OneToMany(mappedBy="department")
    private Set<User> users = new HashSet<User>();
    // 忽略空构造函数和 getter、setter 方法
}
```

我们注意到，Department 实体有一个 users 属性，且类型是 Set，即一个部门拥有多个用户。使用 Set 而不是 List，是因为 Set 结构是存放不同元素的集合，这也是 JPA 要求的。为了体现这种多对一关系，使用了@OneToMany 注解，这里的 One 指的是此注解所在的实体，即 Department 实体，而 Many 是此注解所注解的属性，即 User。

在这种一对多的关系映射上，“One”端采用@OneToMany 注解时必须使用 mappedBy，以声明 Many 端的对象（在这里是 User 实体）的 department 属性提供了对应的关系映射。

### 6.1.4 简化 Entity

现在越来越多的 JPA 应用采用简化的 Entity 定义，去掉了关系映射的相关配置，去掉了数据库外键设置。采用类似 BeetlSQL 或者其他非 ORM 工具，一个表对应了一个简单的对象，这样使用 JPA 变得更加容易。以 User 对象为例子，定义如下：

```
@Entity
public class User  {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)  // 自增长主键
  private Integer id ;
  // 用户名称
  @Column
  private String name ;
  // 创建时间
  @Column(name="create_time")
  private Date createTime ;
  // 部门 ID
  @Column(name="department_id")
  Integer departmentId
}
```

Department 对象则去掉了 users 属性。

```
@Entity
public class Department {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id ;
  @Column
  private String name ;
}
```

这样定义 Entity，不用理解复杂的 JPA 中的 Entity 关系映射概念，JPA 初学者能立即使用 Spring Data JPA。

## 6.2 Repository

Repository 是 Spring Data 的核心概念，抽象了对数据库和 NoSQL 的操作，提供了如下接口供开发者使用：

- CrudRepository，提供了基本的增删改查，批量操作接口。
- PagingAndSortingRepository，集成 CrudRepository，提供了附加的分页查询功能。
- JpaRepository，专门用于 JPA，提供了更多丰富的数据库访问接口，比如根据 Example 来查询，类似 BeetlSQL 的根据模板查询。

> 可以用 Spring Data 操作数据库和 NoSQL 库，NoSQL 包括 Redis、MongoDB、Elasticsearch，我们将在讲解 Elasticsearch 时举例说明。使用 Spring Data 访问 NoSQL 的好处是门槛低，不需掌握 NoSQL 的访问协议，Spring Data 会自动转化为 NoSQL 调用。坏处是 NoSQL 发展过于迅速，Spring Data 未必能支持最新的 NoSQL 版本，另外一个坏处则是对于复杂的 NoSQL 查询、分析等功能，还需要使用 NoSQL 底层 API。因此，实际应用中，还是不可避免地需要使用 NoSQL 自带的访问 API。

### 6.2.1 CrudRepository

Spring Data 提供了 CrudRepository 接口来实现 Entity 的简单增删改查功能。

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {
  /*save 保存 entity，如果 Entity 包含主键，则 Spring Data 认为是更新 entity 操作*/
  <S extends T> S save(S entity);
  /*findById ，根据主键查询实体，返回 Optional 对象.*/
  Optional<T> findById(ID id);
  /*返回所有实体*/
  Iterable<T> findAll();
  /*返回实体个数*/
  Long count();
  /*根据主键删除实体*/
  void deleteById(ID id);
```

```
    /*delete 删除实体*/
    void delete(T entity);

    还有更多方法在此忽略
}
```

T 表示实体类，ID 表示主键类，ID 必须实现序列化接口。比如定义一个 User 的 Repository 接口：

```
public interface UserRepository extends CrudRepository<User, Integer> {

}
```

## 6.2.2 PagingAndSortingRepository

PagingAndSortingRepository 增加了翻页查找和排序相关的操作：

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
    extends CrudRepository<T, ID> {
    /*按照 Sort 指定的排序返回所有实体*/
    Iterable<T> findAll(Sort sort);
    /*Pageable 构造了查询的翻页参数，Page 则表示了查询的具体结果，包含了查询结果集、总
数等*/
    Page<T> findAll(Pageable pageable);
}
```

Sort 类会在 6.2.5 节中说明，Page 和 Pageable 会在 6.2.6 节中进一步说明。

## 6.2.3 JpaRepository

JpaRepository 提供了更多的实用功能，以及通过 Example 对象进行查询。

```
/*返回所有实体*/
List<T> findAll();
/*返回指定一组 id 的所有实体*/
List<T> findAll(Iterable<ID> ids);
/*返回 id 对应的实体，如果未查找，返回空*/
```

```
T getOne(ID id)
/* 保存或者更新一组实体*/
<S extends T> List<S> save(Iterable<S> entities);
/* 查询满足 example 条件的所有对象 */
<S extends T> List<S> findAll(Example<S> example);
/* 查询满足 example 条件的所有对象，并按照 Sort 来排序 */
<S extends T> List<S> findAll(Example<S> example, Sort sort);
```

Example 对象是 Spring Data JPA 提供的用来构造查询条件对象，将在 6.2.10 节 Example 查询中介绍 Example。

### 6.2.4 持久化 Entity

首先创建一个 UserRepository，继承接口 JpaRepository，后续的例子中我们将逐步添加方法，就目前保存 Entity 来说，JpaRepository 已经提供了 save 方法来保存实体。

```
public interface UserRepository extends JpaRepository<User, Integer> {

}
```

Repository 提供 save 方法来保存或者更新一个实体，默认情况下，如果 Entity 主键属性为空，则认为是新的实体，保存实体；反之，如果 Entity 主键属性不为空，则更新实体。

```
@Service
@Transactional
public class UserServiceImpl implements UserService {
  @Autowired
  UserRepository userDao;
  public Integer addUser(User user){
    userDao.save(user);
    Integer id = user.getId();
    user.setName("1"+user.getName());
    userDao.save(user);
    return id;
  }
}
```

addUser 方法直接保存 user 参数，user 此时主键为空，则 save 方法将使用保存操作，插入 entity 到数据库，如果查看控制台，会看到有以下 SQL 语句输出：

```
Hibernate: insert into User (create_time, department_id, name) values
(?, ?, ?)
```

这条 Hibernate 生成的 SQL 语句未包含 id 字段，因为我们在定义实体的时候已经设置 id 策略是数据库自增字段。执行 save 方法后，可以调用 User 的 getId 方法来获取主键。

对于第二个 save 方法，因为 User 实体主键有值，则 Repository 执行更新操作，可以看到控制台有以下输出：

```
Hibernate: update User set create_time=?, department_id=?, name=? where id=?
```

> Spring Data 也提供了另外一个判断实体是否是“新的实体”的方法——Entity 实现 Persistable 接口的 isNew 方法。
>
> Spring Data JPA 默认采用 Hibernate 实现。Hibernate 的 showSql 配置只打印 SQL，并未像 BeetlSQL 那样同时打印 SQL 参数、执行时间等信息，如果需要这些信息，可以使用第三方工具 log4jdbc 来完成。

## 6.2.5 Sort

JpaRepository 提供了如下表所述的内置查询。

| 方　法 | 描　述 |
|---|---|
| List findAll(); | 返回所有实体 |
| List findAll(Iterable ids); | 返回指定 id 的所有实体 |
| T getOne(ID id) | 根据 id 返回对应的实体，如果未找到，则返回空 |
| List findAll(Sort sort); | 返回所有实体，按照指定顺序排序返回 |
| Page findAll(Pageable pageable); | 返回实体列表，实体的 offset 和 limit 通过 pageable 来指定 |

Sort 对象用来指示排序，最简单的 Sort 对象构造可以传入一个属性名列表（不是数据库列名，是属性名），默认采用升序排序。

```
Sort sort = new Sort("id");
return userDao.findAll(sort);
```

控制台会有以下输出：

```
Hibernate: select user0_.id as id1_1_, user0_.create_time as create_t2_1_,
user0_.department_id as departme4_1_, user0_.name as name3_1_ from User user0_
order by user0_.id asc
```

可以看到，Hibernate 根据 Sort 构造了排序条件，Sort("id") 表示按照 id 采用默认的升序进行排序。

其他 Sort 的构造方法还包含：

- public Sort(String... properties)，按照指定的属性列表升序排序。
- public Sort(Direction direction, String... properties)，按照指定属性列表排序，排序由 Direction 指定，Direction 是一个枚举类，有 Direction.ASC 和 Direction.DESC 两类。
- public Sort(Order... orders)，Order 可以通过 Order 静态方法来创建。
  - public static Order asc(String propertyName)；
  - public static Order desc(String propertyName)。

> 上述查询中控制台除了输出 user 对象的查询结果，还输出了多条 department 的查询结果，类似如下：
>
> ```
> Hibernate: select department0_.id as id1_0_0_, department0_.name as
> name2_0_0_ from Department department0_ where department0_.id=?
> ```
>
> 这是因为我们在 User 实体上指定了属性 department 的类型是 Department 对象，且用 @ManyToOne 进行了标示。默认情况下，JPA 还会再发起一次查询以获取 Department 实体。如果系统采用了 6.1.4 节提到的简化 Entity，则 JPA 不会发起对 Department 的查询。

## 6.2.6 Pageable 和 Page

Pageable 接口用于构造翻页查询，PageRequest 是其实现类，可以通过提供的工厂方法创建 PageRequest：

```
public static PageRequest of(int page, int size)
```

也可以在 PageRequest 中加入排序：

```
public static PageRequest of(int page, int size, Sort sort)
```

或者

```
public static PageRequest of(int page, int size, Direction direction,
String... properties)
```

page 总是从 0 开始，表示查询页，size 指每页的期望行数。

Spring Data 翻页查询总是返回 Page 对象，Page 对象提供了以下常用的方法：

- int getTotalPages()，总的页数；
- long getTotalElements()，返回总数；
- List getContent()，返回此次查询的结果集。

```
public List<User> getAllUser(int page,int size) {
  PageRequest pageable = PageRequest.of(page, size);
  Page<User> pageObject =  userDao.findAll(pageable);
  int totalPage = pageObject.getTotalPages();
  long count = pageObject.getTotalElements();
  return pageObject.getContent();
}
```

## 6.2.7 基于方法名字查询

Spring Data 通过查询的方法名和参数名来自动构造一个 JPA OQL 查询，我们可以在 UserRepository 中添加一个查询方法：

```
public interface UserRepository extends JpaRepository<User, Integer> {
    public User findByName(String name);
}
```

方法名和参数名需要遵守一定的规则，Spring Data JPA 才能自动转化为 JPQL：

- 方法名通常包含多个实体属性用于查询，属性之间可以使用 AND 和 OR 连接，也支持 Between、LessThan、GreaterThan、Like；
- 方法名可以以 findBy、getBy、queryBy 开头；
- 查询结果可以排序，方法名包含 OrderBy+属性+ASC(DESC)；
- 可以通过 Top、First 来限定查询结果集；

- 一些特殊的参数可以出现在参数列表里，比如 Pageable、Sort。

以下是一些例子：

```
// 根据名字查询，且按照名字升序
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
// 根据名字查询，且使用翻页查询
Page<User> findByLastname(String lastname, Pageable pageable);
// 查询满足条件的前 10 个用户
List<User> findFirst10ByLastname(String lastname, Sort sort);
// 使用 And 联合查询
List<Person> findByLastnameAndFirstname(String lastname, String firstname);
// 使用 Or 查询
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
String firstname);
// 使用 like 查询，name 必须包含 like 中的%或者?
public User findByNameLike(String name);
```

下表是常用的 Spring Data 支持的关键字。

| 关　键　字 | 例　　子 | 转化的 JPQL 片段 |
|---|---|---|
| And | findByLastnameAndFirstname | ...where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection age) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |

续表

| 关 键 字 | 例 子 | 转化的 JPQL 片段 |
|---|---|---|
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstname) = UPPER(?1) |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |

> **注意**：Spring Data 的 Query 构造既适合本章要重点介绍的 JPA，也适合其他 Spring Data 支持的 NoSQL。在大部分 Spring Boot 应用中，Query 构造都只能创建一些简单的查询。但对于 NoSQL 来说已经足够了，不需要自己再构造 NoSQL 查询。

### 6.2.8 @Query 查询

注解 Query 允许在方法上使用 JPQL。

```
@Query("select u from User u where u.name=?1 and u.department.id=?2")
public User findUser(String name,Integer departmentId);
```

如果你更喜欢 SQL 而不是 JPQL，可以使用@Query 的 nativeQuery 属性，设置为 true：

```
@Query(value="select * from user where name=?1 and department_id=?2",
nativeQuery=true)
public User nativeQuery(String name,Integer departmentId);
```

无论是 JPQL，还是 SQL 语句，都支持“命名参数”：

```
@Query(value="select * from user where name=:name and
department_id=:departmentId",nativeQuery=true)
public User nativeQuery2(String name, Integer departmentId);
```

如果 SQL 或者 JPQL 查询结果集并非 Entity，可以用 Object[]数组代替，比如分组统计每个部门的用户数：

```
@Query(value="select department_id,count(*) from user group by
depatment_id", nativeQuery=true)
public List<Object[]> queryUserCount();
```

这条查询将返回数组，对象类型依赖于查询结果，本例子中，返回的是 String 和 BigInteger 类型。

> **注意：** Spring Data 没有像 BeetlSQL 那样允许将 SQL 查询结果映射到一个任意 POJO 或者 Map 对象上，因此，对于这种**非实体**返回结果，只能用 Object[]数组。数组中每个元素的类型要小心处理，比如 count(*)，返回的类型是 BigInteger，在不同的数据库中，返回的可能是 BigDecimal。

查询时可以使用 Pageable 和 Sort 来对协助“JPQL”完成翻页和排序。

```
@Query(value="select u from User u where u.department.id=?1")
public Page<User> queryUsers(Integer departmentId,Pageable page);
```

@Query 还允许 SQL 更新、删除语句，此时必须搭配@Modifying 使用，比如：

```
@Modifying
@Query("update User u set u.name= ?1 where u.id= ?2")
int updateName(String name,Integer id);
```

## 6.2.9 使用 JPA Query

上面提到的 Repository 查询类似 BeetlSQL 的 BaseMapper 接口，提供了内置查询或者注解查询。通常情况下，JPQL 语句是运行时决定的，比如查询，需要根据输入条件来组成不同的 JPQL 语句，这时候，必须使用底层的 EntityManager 来完成查询。EntityManager 是 JPA 提供的数据库访问接口，类似 BeetlSQL 的 SQLManager。

EntityManager 提供了实体操作的所有接口，可以通过自动注入方式注入到 Spring 管理的 Bean 中，通常是由@Service 注解的业务处理类上。本节不打算介绍 EntityManager，会提一下在复杂查询中如何使用 EntityManager 以补充 Repository 的不足。

```
@Service
@Transactional
public class UserServiceImpl implements UserService {

  @Autowired
  EntityManager em;
}
```

EntityManager 提供了 createQuery(String jpqlString)来创建 Query 对象。Query 对象提供了数据的查询和翻页功能。较为复杂的动态翻页查询大概有以下四部分结构：

- 构造一个 JPQL，包含 JPQL 的条件查询部分，将复用这个 JPQA 作为求总数和翻页查询的基础 JPQL；
- 用基础的 JPQL 构造一个查询符合条件的总数，调用 Query.getSingleResult 查询出满足条件的总数；
- 用基础的 JPQL 构造结果查询，并调用 Query.getResultList 查询出结果集；
- 组装成一个 Spring Data 的 Page 实例，返回。

```
public Page<User> queryUser2(Integer departmentId,Pageable page){
  // 构造 JPQL 和实际的参数
  StringBuilder baseJpql = new StringBuilder("from User u where 1=1 ");
  Map<String,Object> paras = new HashMap<String,Object>();
  if(departmentId!=null){
    baseJpql.append("and  u.department.id=:deptId");
    paras.put("deptId", departmentId);
  }
  // 查询满足条件的总数
  long count = getQueryCount(baseJpql,paras);
  if(count==0){
    return new PageImpl(Collections.emptyList(),page,0);
  }
  // 查询满足条件的结果集
  List list = getQueryResult(baseJpql,paras,page);
  // 返回结果
  Page ret = new PageImpl(list,page,count);
  return ret;
}
```

代码段第一部分用来构造一个基础的 JPQL 语句，通过对输入参数的判断，从而构造整个 JPQL，以及 JPQL 需要的参数。

getQueryCount 的代码如下：

```
private Long getQueryCount(StringBuilder baseJpql,Map<String,Object> paras){
  Query query= em.createQuery("select count(1) "+baseJpql.toString());
```

```
    setQueryParameter(query,paras);
    Number number = (Number)query.getSingleResult();
    return number.longValue();
  }
```

getQueryCount 构造了一个查询总数的 JPQL，并调用 setQueryParameter 来设置 Query 对象需要的参数，最后调用 getSingleResult 来获取查询结果总数。由于我们并不知道返回的数据类型（在 MySQL 例子中，实际上返回的是 BigInteger），因此代码先强制转化为 Number，然后转化为 Long 类型。对于 Spring Boot 应用来说，Long 精度已经足够。

setQueryParameter 是一个通用的设置 Query 对象需要的参数的方法，片段如下：

```
  private void setQueryParameter(Query query,Map<String,Object> paras ){
    for(Entry<String,Object> entry:paras.entrySet()){
      query.setParameter(entry.getKey(),entry.getValue());
    }
  }
```

本例中，paras 包含了 deptId。

当返回的 count>0 时，会调用 getQueryResult 方法，用于查询结果集，代码片段如下：

```
  private  List  getQueryResult(StringBuilder  baseJpql,Map<String,Object>
paras,Pageable page){
    Query query= em.createQuery("select u "+baseJpql.toString());
    setQueryParameter(query,paras);
    query.setFirstResult(page.getOffset());
    query.setMaxResults(page.getPageNumber());
    List list = query.getResultList();
    return list;
  }
```

Query 对象在查询结果集的时候，提供 setFirstResult 来设置查询的起始位置，setMaxResults 来设置此次查询期望返回的总数。这两个参数正好对应 page 对象的 getOffset 和 getPageNumber。

如果观察控制台输出，会发现输出了以下 SQL：

```
  Hibernate:  select  count(1)  as  col_0_0_  from  User  user0_  where  1=1  and
user0_.department_id=?
  Hibernate: select user0_.id as id1_1_, user0_.create_time as create_t2_1_,
user0_.department_id as departme4_1_, user0_.name as name3_1_ from User user0_
```

```
where 1=1 and user0_.department_id=?
```

第一条用于查询总数，第二条用于查询结果集。

> EntityManager 也支持直接使用 Native SQL。可以调用 em.createNativeQuery(sqlString)，用法同本节的例子一致。

## 6.2.10 Example 查询

我们在 6.2.7 节中，通过方法名来构造 JPQL，在 6.2.8 节和 6.2.9 节中，直接使用 JPQL 来进行查询，本节介绍的 Example 对象则是一种折中方案，它允许根据实体创建一个 Example 对象，Spring Data 通过 Example 对象来构造 JPQL。

```
public List<User> getByExample(String name) {
  User user = new User();
  Department dept = new Department();
  user.setName(name);
  dept.setId(1);
  user.setDepartment(dept);
  Example<User> example = Example.of(user);
  List<User> list = userDao.findAll(example);
  return list;
}
```

以上代码首先创建了 User 对象，设置了查询条件，名称为参数 name，部门 id 为 1，通过 Example.of 构建了此查询。

userDao 接口继承了 JpaRespository，因此提供 findAll 接口来查询出所有满足 example 的实体。

大部分查询都并非完全匹配查询，ExampleMatcher 提供了更多的条件指定。比如以 name 开头的所有用户，则可以用以下代码构造：

```
  ExampleMatcher matcher = ExampleMatcher.matching()
                  .withMatcher("name",
GenericPropertyMatchers.startsWith().ignoreCase());
  Example<User> example = Example.of(user, matcher);
```

7 chapter

# 第 7 章 Spring Boot 配置

在 Spring Boot 出现之前，Spring 项目会存在多个配置文件，比如 web.xml，配置 Spring 的多个 application-xxx.xml，xxx 代表配置 Spring 的某一个功能，如 application-datasource.xml、application-mvc.xml。应用自身也需要多个配置文件，还需要编写代码去读取这些配置文件的参数。现在 Spring Boot 简化了 Spring 配置的管理和读取，只需要一个 application.properties，并提供了多种读取配置文件的方式。

## 7.1 配置 Spring Boot

Spring Boot 默认启动的是 8080 端口，Web 上下文是“/”，可以通过配置 application.properties 来重新配置 Spring Boot。

### 7.1.1 服务器配置

如果想更换其他端口，需要配置属性 server.port，比如在 application.properties 中输入如下代码：

```
server.port=9090
```

也可以在命令行中指定启动端口，比如传入参数--server.port=9000：

```
java -jar bootsample.jar --server.port=9000
```

或者传入虚拟机系统属性：

```
java -Dserver.port=9000 -jar bootsample.jar
```

> 以上三种方式都可以指定 Web 监听端口，Spring Boot 的所有配置属性也都支持这三种方式，一般而言，传入虚拟机系统属性较为适用。
>
> 在一台机器上部署多份 Spring Boot 应用有很多好处，比如，提升系统吞吐量和性能，防止误操作关掉某一台应用，或者是在应用升级的时候，可以逐个升级而不影响系统服务。为了实现这个功能，只需要配置不同的监听端口就可以了。

Spring Boot 默认为应用配置的上下文访问目录是“/”，可以通过配置文件或者命令行，配置 server.context-path：

```
server.servlet.Path=/config
```

常用的服务器配置的属性如下表所示。

| 属　性 | 描　述 |
|---|---|
| server.address | 服务器 IP 绑定地址，如果你的主机上有多个网卡，可以绑定一个 IP 地址 |
| server.session.timeout | 会话过期时间，以秒为单位 |
| server.error.path | 服务器出错后的处理路径/error，我们在第 4 章已经介绍过如何处理异常 |
| server.servlet contextpath | Spring Boot 应用的上下文 |
| server.port | Spring Boot 应用监听端口 |

## 7.1.2 使用其他 Web 服务器

Spring Boot 内置了 Tomcat，同时还支持 Jetty、Undertow 作为 Web 服务器。使用这些应用服务器，只需要引入相应的 starter。

Undertow

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
```

```
</dependency>
```

Jetty

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

同时，还需要在 Spring-boot-starter-web 中去除 Tomcat 依赖：

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
   <exclusions>
      <exclusion>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
   </exclusions>
</dependency>
```

> 到目前为止，Undertow 的性能要优于 Tomcat 和 Jetty，推荐使用 Undertow 作为 Spring Boot 应用服务器。
> 以上配置参考 Spring Boot 官网附录 A:Common application properties。

server.tomcat.*包含了 Tomcat 的相关配置，较为重要的配置如下：

```
# 打开 Tomcat 访问日志
server.tomcat.accesslog.enabled=false
# 访问日志所在的目录
server.tomcat.accesslog.directory=logs
# 允许 HTTP 请求缓存到请求队列的最大个数，默认不限制
server.tomcat.accept-count=
# 最大连接数，默认不限制，如果一旦连接数到达，剩下的连接将会保存到请求缓存队列里，也就是 accept-count 指定队列
server.tomcat.max-connections=
# 最大工作线程数
```

```
server.tomcat.max-threads=
#HTTP POST 内容最大长度，默认不限制
server.tomcat.max-http-post-size
```

server.undertow.* 包含了 undertow 相关配置，较为重要的配置如下：

```
# 打开 undertow 日志，默认为 false
server.undertow.accesslog.enabled=false
# 访问日志所在目录
server.undertow.accesslog.dir=logs
# 创建工作线程的 I/O 线程，默认为 2 或者 CPU 的个数
server.undertow.io-threads=
# 工作线程个数，默认为 I/O 线程个数的 8 倍
server.undertow.worker-threads
# HTTP POST 内容最大长度，默认不限制
server.undertow.max-http-post-size=0
```

server.jetty.* 包含了 Jetty 相关配置，较为重要的配置如下：

```
# 打开 Jetty 日志，默认为 false
server.jetty.accesslog.enabled=false
# 访问日志所在目录
server.jetty.accesslog.dir=logs
# acceptors 线程个数，用来接受访问请求，相当于工作线程
server.jetty.acceptors=
# selectors 线程个数，和 CPU 个数相关，默认是可用(CPU+1)/2，用于分配请求给工作线程
server.jetty.selectors=
```

## 7.1.3 配置启动信息

Spring Boot 启动的欢迎信息也可以进行配置，默认启动后，控制台打印“Spring”：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
```

可以在 classpath 中增加 banner.txt，显示你自己的输出信息，在 Spring Boot 项目的 resources 目录下新建一个 banner.txt，内容如下：

```
######Hello,Config Springboot
```

再次运行 Spring Boot，会发现启动信息已经改变。

以笔者的 banner.txt 为例，输出如下：

```
////////////////////////////////////////////////////////////////////
//                          _ooOoo_                               //
//                         o8888888o                              //
//                         88" . "88                              //
//                         (| ^_^ |)                              //
//                         O\  =  /O                              //
//                      ____/`---'\____                           //
//                    .'  \\|     |//  `.                         //
//                   /  \\|||  :  |||//  \                        //
//                  /  _||||| -:- |||||-  \                       //
//                  |   | \\\  -  /// |   |                       //
//                  | \_|  ''\---/''  |   |                       //
//                  \  .-\__  `-`  ___/-. /                       //
//                ___`. .'  /--.--\  `. . ___                     //
//              ."" '<  `.___\_<|>_/___.'  >'"".                  //
//            | | :  `- \`.;`\ _ /`;.`/ - ` : | |                 //
//            \  \ `-.   \_ __\ /__ _/   .-` /  /                 //
//      ========`-.____`-.___\_____/___.-`____.-'========         //
//                           `=---='                              //
//      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^      //
//            佛祖保佑       永不宕机      永无BUG                //
////////////////////////////////////////////////////////////////////
```

也可以设置 banner.gif(png,jpg)，控制台自动将图片转为 ASCII 字符，作为启动信息输出，比如公司的 Logo，将图片复制到 resources 目录下即可。

以下是 application.properties 的默认配置：

```
banner.charset=UTF-8 # banner.txt 字符集
```

```
banner.location=classpath:banner.txt
banner.image.location=classpath:banner.gif # 如果使用图片，图片的位置可以使用
jpg/png
banner.image.width= 76 # 图片宽度，这里指转为字符的个数，越多越清楚
banner.image.height=76 # 图片长度
banner.image.margin= 2 # 图片与左边的边距，默认为 2 个字符
```

### 7.1.4 配置浏览器显示 ico

Spring Boot 的 webapp 启动后，通过浏览器访问，浏览器上会显示一个绿色的树叶的图标，那是 Spring Boot 的官方 Logo。如果需要更换成自己的图标，在项目的 resources 目录下新建一个 static 目录，在 static 目录下创建 images 目录（或者任意放置图片的目录），然后将项目的 favicon.ico 放在 images 目录下，每个页面添加以下样式即可：

```
<link rel="shortcut icon" href="/images/apple.ico">
```

## 7.2 日志配置

默认情况下，不需要对日志做任何配置就可以使用，Spring Boot 使用 LogBack 作为日志的实现，使用 apache Commons Logging 作为日志接口，因此代码中通常是这样的：

```
public class HelloworldController {
  private Log log = LogFactory.getLog(HelloworldController.class);
  ...
}
```

日志格式类似这样：

```
2017-03-14 17:10:40.253 INFO 6396 --- [ restartedMain]
com.coamc.starter.CosonleApplication      : Starting ...
2017-03-14 17:10:40.255 INFO 6396 --- [ restartedMain]
com.coamc.starter.CosonleApplication      : No active profile set, ....
```

日志每行内容的格式如下：

- 日期和时间；

- 日志级别，有 ERROR、WARN、INFO、DEBUG 和 TRACE；
- 进程 id，Spring Boot 应用的进程 id；
- ---，分隔符号，后面是日志消息；
- [xxx]，线程的名称；
- 类名；
- 消息体。

默认情况下，INFO 级别以上的信息才会打印到控制台，可以自己设定日志输出级别，比如在 application.properties 中加入以下代码：

```
logging.level.root=info
# org 包下的日志级别
logging.level.org=warn
logging.level.com.yourcorp=debug
```

指定默认的级别是 INFO，但包名是 org 开头的类，日志级别是 WARN，org 包名的类大多是第三方依赖库，有时候没有必要显示 INFO 级别，com.yourcorp 开头的类使用 debug。

Spring Boot 默认并未输出日志到文件，可以在 application.properties 中指定日志输出：

```
logging.file = my.log
```

日志输出到 my.log 中，位于 Spring Boot 应用运行的当前目录，比如项目工程目录下，也可以指定日志存放的路径，使用：

```
logging.path=e:/temp/log
```

这样，会默认在 e:/temp/log 下生成一个叫 spring.log 的日志文件。

无论用哪种方式记录日志文件，当日志文件到达 10MB 的时候，会自动重新生成一个新日志文件。

Spring Boot 支持对控制台日志输出和文件输出进行格式控制，代码如下（仅使用内置的 logback）：

```
logging.pattern.console=%level %date{HH:mm:ss} %logger{20}.%M %L :%m%n
logging.pattern.file= %level %date{ISO8601} [%thread] %logger{20}.%M %L :%m%n
```

- %level，表示输出日志级别。

- %date，表示日志发生时的时间，HH:mm:ss 输出时分秒，比较合适用于控制台查看，ISO8601 则是标准日期输出，相当于 yyyy-MM-dd HH:mm:ss.SSS。
- %logger，用于输出 Logger 的名字，包名+类名，{n}限定了输出长度，如果输出长度不够，尽可能显示类名、压缩包名。
- %thread，当前线程名。
- %M，日志发生时的方法名字。
- %L，日志调用所在代码行，线上运行时不建议使用此参数，因为获取代码行对性能有消耗。
- %m，日志消息。
- %n，日志换行。

> Spring Boot 支持多种日志框架，如 Log4J2、Logback、Java Util Logging 等，但建议使用内置的 Logback 即可。
>
> 也可以通过在 resources 目录下使用 logback.xml 或者 logback-spring.xml 来对 Logback 进行更详细的配置。

我们可以通过命令行的方式来改变日志配置，也可以通过 Actuator 功能在运行时动态改变日志配置，关于 Actuator，在后面会专门讲述。以下是通过参数--debug 将系统日志级别调整为 debug 方式。

```
java -jar myapp.jar -debug
```

或者传入一个特定的日志配置：

```
java -jar myapp.jar --logging.level.com.bee.sample=error
```

## 7.3 读取应用配置

可以在应用中读取 application.properties 文件，Spring Boot 提供了三种方式，通用的 Enviroment 类，可以通过 key-value 方式获取到 application.properties 中的值，也可以通过@Value 注解，自动注入属性值，还可以将一组属性自动注入到一个配置类中。

## 7.3.1 Environment

Environment是一个通用的读取应用程序运行时的环境变量的类，可以读取application.properties、命令行输入参数、系统属性、操作系统环境变量等。可以通过Spring 容器自动注入，比如在Spring管理的Bean中：

```
@Configuration
public class EnvConfig {
  @Autowired private  Environment env;
  public int getServerPort(){
      return env.getProperty("server.port", Integer.class);
  }
}
```

下表是一些读取的例子。

| 读　　取 | 返　回　值 |
|---|---|
| env.getProperty("user.dir") | 程序运行的目录，如果在IDE中运行，就是工程目录，user.dir是系统属性 |
| env.getProperty("user.home") | 执行程序的用户的home目录，user.home是系统属性 |
| env.getProperty("JAVA_HOME") | 读取设置的环境变量（不区分大小写） |
| env.getProperty("server.port") | 读取server.port，来自application.properties |

> Environment 是 Spring Boot 最早初始化的一个类，因此可以用在 Spring 应用的任何地方。

## 7.3.2 @Value

直接通过@Value注解注入一个配置信息到Spring管理的Bean中：

```
@RequestMapping("/showvalue.html")
public @ResponseBody String value(@Value("${server.port}")  int port){
return "port:"+port;
}
```

> **注意：**
> @Value 并不能在任何 Spring 管理的 Bean 中使用，因为@Value 本身是通过 AutowiredAnnotationBeanPostProcessor 实现的，它是 BeanPostProcessor 接口的实现类，因此任何 BeanPostProcessor 和 BeanFactoryPostProcessor 的子类中都不能使用@Value 来注入属性，因为那时候@Value 还没有被处理。

@Value 注解支持 SpEL 表达式，如果属性不存在，可以为其提供一个默认值：

```
@Value("${cache.enable:false}")
private boolean isCache;
```

## 7.3.3 @ConfigurationProperties

通常情况下，将一组同样类型的配置属性映射为一个类更为方便，比如服务器配置，在 application.properties 中写成如下配置：

```
server.port=9090
server.context-path=/config
```

以上三个配置属性都与 Web 服务器配置相关，都有 server 前缀，因此可以使用注解 @ConfigurationProperties 来获取这一组实现，代码如下：

```
@ConfigurationProperties("server")
@Configuration
public class ServerConfig {
  private int port;
  private String contextPath;
  public int getPort() {
    return port;
  }
  public void setPort(int port) {
    this.port = port;
  }
  public String getContextPath() {
    return contextPath;
  }
  public void setContextPath(String contextPath) {
```

```
    this.contextPath = contextPath;
  }

}
```

在处理 ConfigurationProperties 注解的类的时候，自动会将“-”或者“_”去掉，转化为 Java 命名规范，如将 context-path 转为 contextPath。

> ConfigurationProperties 和@Value 的功能差不多，建议使用@ConfigurationProperties，因为它能将一组属性统一管理。@Value 的优点是支持 SpEL 表达式，但 SpEL 表达式是把双刃剑，不容易调试和重构。

## 7.4 Spring Boot 自动装配

Spring 提供了注解@Configuration，用来配置多个 Bean。我们在前面章节中已经看到通过@Configuration 来配置数据源等，这一节继续学习 Configuration 和自动装配。

### 7.4.1 @Configuration 和@Bean

Spring 的 Java 配置的核心就是使用@Configuration 作用在类上，并且联合在此类上多个用@Bean 注解的方法，声明 Spring 管理的 Bean。它类似较早的 XML 的配置方式：

```
<beans>
  <bean id="testBean" class="xxx.TestBean"></bean>
</beans>
```

采用@Configuration，类似如下配置：

```
@Configuration
public class MyConfiguration {

  @Bean("testBean")
  public TestBean getBean(){
    return new TestBean();
  }

}
```

以上代码中，MyConfiguration 类使用了注解@Configuration，向 Spring 表明这是一个配置类，类里的所有带@Bean 注解的方法都会被 Spring 调用，返回对象将会作为一个 Spring 容器管理的 Bean。注解@Bean 可以给 Bean 指定一个名字，比如此例中的“testBean”，Bean 的名字通常是类名首字母小写方式，如果不指定名字，Spring 将用方法名作为 Bean 的名称。

通常配置类还需要获取外部属性（配置文件、系统变量、环境变量），Environment 提供了获取这些外部属性的 API。

可以在@Bean 注解的方法上提供任意参数来说明依赖，比如 MyService 需要依赖数据源 datasource：

```
@Bean
public MyService getMyService(DataSource datasource) {
  return new MyService(datasource);
}
```

datasource 是我们已经在 Spring Boot 其他地方配置好的数据源。

在配置好 Bean 后，可以通过@Autowired 在任何地方自动注入，比如某业务代码中：

```
@Service
public class Service{
  @Autowired MyService myService;
}
```

## 7.4.2 Bean 条件装配

Spring Boot 可以通过有无指定 Bean 来决定是否配置 Bean，使用@ConditionalOnBean，在当前上下文中存在某个对象时，才会实例化一个 Bean；使用@ConditionalOnMissingBean，在当前上下文中不存在某个对象时，才会实例化一个 Bean。

```
@Configuration
@ConditionalOnBean(DataSource.class)
public class BeetlSqlConfig {
}
```

BeetlSqlConfig 配置生效的前提是上下文中已经配置了 DataSource。

```
@Configuration
```

```
public class MyAutoConfiguration {
  @Bean
  @ConditionalOnMissingBean
  public MyService myService() { ... }
}
```

如果没有配置过 MyService，则调用 myService。

## 7.4.3 Class 条件装配

Class 条件装配是按照某个类是否在 Classpath 中来决定是否要配置 Bean。@ConditionalOnClass 表示当 classpath 有指定的类时，配置生效，代码如下：

```
@Configuration
@ConditionalOnClass(JestClient.class)
public class JestAutoConfiguration {
}
```

这段代码用于配置 Elasticsearch，使用 Jest 驱动，因此配置生效的前提条件是 classpath 有 JestClient.class 类。

@ConditionalOnMissingClass 表示当 classpath 中没有指定的类的时候，配置生效。

## 7.4.4 Environment 装配

可以根据 Spring Boot 的 Environment 属性来决定配置是否生效：

```
@ConditionalOnProperty(name = "message.center.enabled", havingValue =
"true", matchIfMissing = true)
public class MessageCenterAutoConfiguration {


}
```

@ConditionalOnProperty 注解根据 name 来读取 Spring Boot 的 Environment 的变量包含的属性，根据其值与 havingValue 的值比较结果决定配置是否生效。如果没有指定 havingValue，只要属性不为 false，配置都能生效。

matchIfMissing 为 true 意味着如果 Environment 没有包含“message.center.enabled”，配置

也能生效，默认为 false。

### 7.4.5 其他条件装配

- @ConditionalOnExpression，当表达式为 true 时，才会实例化一个 Bean，支持 SpEL 表达式，比如根据配置文件中的某个值来决定配置是否生效。
- ConditionalOnJava，当存在指定的 Java 版本的时候。

```
@ConditionalOnJava(range=Range.EQUAL_OR_NEWER,value=JavaVersion.EIGHT)
```

### 7.4.6 联合多个条件

前面讲的注解可同@Configuration 联合使用，如果满足条件，则其类下所有配置的 Bean 都可以使用，也可以单独同@Bean 使用。

以 Spring Boot Cache 为例（支持多种缓存实现，如 SimpleCache），可以用 HashMap 实现，还可以是分布式的 Redis 实现方式。无论是哪种实现方式，只需要配置好 CacheManager——缓存的抽象管理类。对应于 SimpleCache，是 ConcurrentMapCacheManager 实现；对应于 Redis 缓存，是 RedisCacheManager 实现。

以下类是 SimpleCacheConfiguration 的自动装配实现。

```
@Configuration
@ConditionalOnMissingBean(CacheManager.class)
@Conditional(CacheCondition.class)
class SimpleCacheConfiguration {
  @Bean
  public ConcurrentMapCacheManager cacheManager() {
    ConcurrentMapCacheManager cacheManager = new ConcurrentMapCacheManager();
    // 忽略其他配置代码
    return cacheManager
  }
}
```

注解@Configuration 向 Spring 表明这是一个配置表，注解@ConditionalOnMissingBean 可以接受一个或者多个类作为参数，如果还没有配置过 CacheManager，则配置类可以生效。注解@Conditional 是一个更为通用的条件配置类，其后参数的实例实现了 Condition 接口的 match 方

法，我们将在后面讲解。

RedisCacheConfiguration 的配置类似 SimpleCacheConfiguration，唯一不同的地方在于需要判断系统是否有 RedisTemplate 类——一个用来访问 Redis 较为底层的API，我们将在 Redis 一章中了解更详细的内容。RedisCacheConfiguration 的配置如下：

```
@Configuration
@AutoConfigureAfter(RedisAutoConfiguration.class)
@ConditionalOnBean(RedisTemplate.class)
@ConditionalOnMissingBean(CacheManager.class)
@Conditional(CacheCondition.class)
class RedisCacheConfiguration {
  @Bean
  public RedisCacheManager cacheManager(RedisTemplate<Object, Object>
redisTemplate) {
    RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
    // 忽略其他代码
    return cacheManager;
  }
}
```

Redis 的缓存配置需要确保 RedisTemplate 已经配置，因此使用了注解@AutoConfigureAfter，表示此配置类需要在 RedisAutoConfiguration 配置类后再生效，ConditionalOnBean 则表示如果成功配置好 RedisTemplate，此配置才能继续生效。

## 7.4.7 Condition 接口

当 Spring Boot 内置的 ConditionalOnBean、ConditionalOnClass 等无法满足需要的时候，可以自己构造一个 Condition 实现，使用注解@Conditional 来引用此 Condition 实现。

Condition 接口的定义如下：

```
public interface Condition {
  boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);
}
```

ConditionContext 类可以得到用于帮助条件判断的辅助类：

- Environment，可以读取系统属性、环境变量、配置参数等来作为判断条件，比如当配

置文件中某一项存在的时候配置才生效。

- ResourceLoader，一个 Spring 类，用来加载和判断资源文件，比如当某个配置文件存在时配置才生效。
- ConfigurableListableBeanFactory，Spring 容器。

以下配置了一个对存入数据库的用户手机进行加密的类，使用了@Conditional 注解，要求当存在 salt.txt 文件且配置允许手机加密时才生效。

```
@Configuration

public class MobileEncryptCondtion {
  @Bean
  @Conditional(EncryptCondition.class)
  public MobileEncryptBean mobileEncryptBean(){
    return new MobileEncryptBean();
  }

  static class EncryptCondition implements Condition{

    public boolean matches(ConditionContext ctx, AnnotatedTypeMetadata metadata) {
      Resource res = ctx.getResourceLoader().getResource("salt.txt");
      Environment env = ctx.getEnvironment();
      return res.exists()&&env.containsProperty("mobile.encrypt.enable");
    }

  }
}
```

我们在前面看到的@ConditionalOnMissingBean、@ConditionalOnJava 等注解实际上也是通过@Conditional 完成的，以@ConditionalOnJava 为例：

```
@Conditional(OnJavaCondition.class)
public @interface ConditionalOnJava {
  Range range() default Range.EQUAL_OR_NEWER;
  JavaVersion value();
}
```

满足 ConditionalOnJava 的条件是在 OnJavaCondition 类中实现的。OnJavaCondition 类在这里就

不详细说明了，其关键代码使用了 AnnotatedTypeMetadata 来获取@ConditionalOnJava 注解的属性信息，比如 value()方法返回的 String[]：

```
Map<String, Object> attributes = metadata
          .getAnnotationAttributes(ConditionalOnJava.class.getName());
Range range = (Range) attributes.get("range");
JavaVersion version = (JavaVersion) attributes.get("value");
```

### 7.4.8 制作 Starter

到目前为止，我们已经使用了很多 Starter，最常用的是 spring-boot-starter-web，还有 spring-boot-devtools，也在视图和数据库访问中使用了 beetl-framework-starter。starter 包含了两项主要的功能才使得 Spring Boot 变得非常容易使用：

- 配置了依赖库，如 spring-boot-starter-web 依赖了 Tomcat 等，beetl-framework-starter 依赖了 Beetl 和 BeetlSQL 等。
- 自动配置，在前面几节已经讲述了子自动配置的知识。

编写 Starter 与编写一个普通的 Spring Boot 应用没有太大区别，只需要在 pom 文件中写好依赖，使用@Configuration 和@Bean 来自动装配集成。唯一的一点区别是需要告诉 Spring Boot 自动装配类在哪里，这是通过 spring.factories 文件完成的。

spring.factories 文件位于 resources/META-INF 目录下，以 beetl-framework-starter 为例：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.ibeetl
.starter.BeetlTemplateConfig,com.ibeetl.starter.BeetlSqlConfig
```

org.springframework.boot.autoconfigure.EnableAutoConfiguration 后面的类名说明了自动装配类，如果有多个，则用逗号分开。

将 Spring Boot 应用导出成 jar 文件并通过 Maven 发布到仓库，这样就具备了一个 Starter。

# 第8章 部署Spring Boot应用

前面 7 章讲述了使用 Spring Boot 极速开发一个 Web 应用系统，这一章我们讲述如何部署 Spring Boot 应用，Spring Boot 可以以 jar 方式运行，也可以部署到支持 Servlet3.0 或者支持较早的 Servlet2.5 的 Web 服务器上。

Spring Boot 应用部署通常会面临多个部署环境，如测试环境、线上环境、演示环境等，本章也会讲述如何支持 Spring Boot 多环境部署。

## 8.1 以 jar 文件运行

Spring Boot 默认以 jar 包方式运行，可以在 Maven 配置如下插件，将 Spring Boot 导出成可执行的 jar 文件。

```
<build>
  <plugins>
    .......
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <executable>true</executable>
```

```
        </configuration>
      </plugin>
    </plugins>
  </build>
```

在工程目录下的运行命令行中运行 mvn package：

```
>mvn package
```

package 会将 Maven 工程打包成一个可执行的 jar 文件存放在 target 目录下，在控制台中看到有如下输出则表示输出成功：

```
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ ch8.deploy ---
[INFO] Building jar: /Users/xiandafu/git/sb2/8_deploy/code/ch8.deploy/
target/ch8.deploy-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.0.M3:repackage (default) @
ch8.deploy ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.264 s
[INFO] Finished at: 2017-06-18T12:08:29+08:00
[INFO] Final Memory: 20M/227M
[INFO] ------------------------------------------------------------------------
```

> **注意**：在编写本章的时候，使用的 Spring Boot 版本是 2.0.0.M3。

为了验证打包成功，可以以 jar 方式运行打包好的 ch8.deploy-0.0.1-SNAPSHOT.jar：

```
>java -jar target/target/ch8.deploy-0.0.1-SNAPSHOT.jar:
```

可以将此 jar 文件部署到服务器上，以 jar 方式运行。通常情况下，还需要指定服务器端口、数据库连接地址等信息，我们将在多环境部署中详细讲解。简单的方法是在上面的命令行中传入系统属性，比如，如果应用使用的端口是 8000，数据库的密码是 Test123!（假设数据库用户名在多环境中保持统一）：

```
>java -jar -Dserver.port=9000 -Dspring.datasource.password=Test123!
target/ch8.deploy-0.0.1-SNAPSHOT.jar
```

> 注意，另外一种传入配置的方法是使用命令行参数的方式传递：
>
> java -jar target/target/ch8.deploy-0.0.1-SNAPSHOT.jar --server.port=9000 --spring.datasource.password= Test123!

## 8.2 以 war 方式部署

Spring Boot 默认自带了一个嵌入式的 Tomcat 服务器，可以以 jar 方式运行，更为常见的情况是需要将 Spring Boot 应用打成一个 war 包，部署到 Tomcat 、Jetty 服务器，或者商业的 Weblogic、Websphere 上。这种情况下，需要将 pom 中的 packaging 改成 war 方式：

```
<?xml version="1.0" encoding="UTF-8"?>
<project .....>
  <artifactId>ch8.deploy</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
</project>
```

需要将嵌入的 Tomcat 依赖方式改成 provided，因此增加如下依赖说明：

```
<dependencies>
  <!-- … -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
  </dependency>
  <!-- … -->
</dependencies>
```

还需要对 Main 类进行修改，使其继承 SpringBootServletInitializer 类，并重载 configure 方法，代码如下：

```
@SpringBootApplication
public class Ch8Application extends SpringBootServletInitializer {
  @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
```

```
application){
        return application.sources(Ch8Application.class);
    }
    public static void main(String[] args) {
        SpringApplication.run(Ch8Application.class, args);
    }
}
```

以上修改完毕后，就可以再次运行 mvn package，打包成一个 war 包：

```
[INFO] Packaging webapp
[INFO] Assembling webapp [ch8.deploy] in [/Users/xiandafu/git/sb2/8_deploy/
code/ch8.deploy/target/ch8.deploy-0.0.1-SNAPSHOT]
[INFO] Processing war project
[INFO] Webapp assembled in [170 msecs]
[INFO] Building war: /Users/xiandafu/git/sb2/8_deploy/code/ch8.deploy/
target/ch8.deploy-0.0.1-SNAPSHOT.war
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.0.M3:repackage (default) @
ch8.deploy ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

> **注意**：这种部署方式适合 Servlet3.0 的容器，对于 Tomcat7、Jetty9、Undertow1.2 以上均支持。

如果你的 Web 服务器是 Weblogic，还需要一些额外的方法，首先程序入口需要完成 WebApplicationInitializer 接口，类似如下：

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements
WebApplicationInitializer {
```

```
}
```

其次，要当心 Spring Boot 用的 jar 包与 Weblogic 的冲突，这种情况下，可以在 WEB-INF 目录下增加 weblogic.xml，配置优先使用应用自带的 jar 包。比如，应用的日志系统优先采用 logback：

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
  xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
    http://xmlns.oracle.com/weblogic/weblogic-web-app
    http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-
web-app.xsd">
  <wls:container-descriptor>
    <wls:prefer-application-packages>
      <wls:package-name>org.slf4j</wls:package-name>
    </wls:prefer-application-packages>
  </wls:container-descriptor>
</wls:weblogic-web-app>
```

## 8.3 多环境部署

再简单的应用系统，通常都有两个环境——开发环境和线上环境。大型的企业应用还会有更多的环境，比如测试环境、准线上环境、演示环境等。应用的版本也可能对应了多个环境，比如 1.0 版本的演示环境、2.0 版本的演示环境。这些环境对应了不同的配置要求，通常有：

- 数据库的访问地址不同，数据库用户名和密码也不同，连接池的配置也大不一样，开发环境可能只配置 5 个连接，而线上环境则多达数百个。
- 日志配置不同，开发和测试环境的日志级别可能是 debug 级别，日志信息也常常包含代码所在行，而线上环境则通常是 INFO 级别，且为了性能考虑不会输出日志所在行。另外，对日志文件要求也不尽相同，线上环境要求日志文件能定时打包压缩，开发环境则往往不会配置日志文件。
- 访问的第三方系统不一样，复杂的应用通常要访问第三方系统，如系统内的 Redis 缓存，通过 REST 访问其他 Spring Boot 应用，这都需要配置不同的 IP 地址。

- 不同的环境有不同的开关属性，比如开发系统，需要关闭短信、微信的通知功能，而演示环境、线上环境则需要打开这些配置。

我们知道 Spring Boot 应用简化了配置，所有重要的配置属性都放在 application.properties 中。Spring Boot 允许准备多个配置文件，可以在系统部署的时候，指定使用哪个配置文件覆盖默认的 application.properties，从而完成多环境部署。

需要在 resources 下创建 application-{profile}.properties 的配置文件，其中 profile 可以是任意名字，比如：

- test，表示测试环境；
- prod，表示线上环境；
- pre-prod，预发布环境；
- demo1.0，1.0 版本演示环境。

这些配置文件可以添加或者覆盖 application.properties 文件的属性。

在环境变量中，spring.profiles.active 指定使用哪个 profile，比如：

```
java -jar -Dspring.profiles.active=prod target/ch8.deploy-0.0.1-SNAPSHOT.jar
```

以上配置启动后，Spring Boot 将读取 resources/application-prod.properties 配置文件，覆盖默认的 application.properties 选项。

application.properties 的内容如下：

```
server.port=8080
......
```

application-prod.properties 的内容如下：

```
server.port=9000
......
```

如果使用 war 方式部署，添加系统属性是比较好的方式，下面以 Tomcat 为例进行说明。

编辑 catalina.sh，在 sh 文件的开头部分添加如下内容：

```
JAVA_OPTS="-Dspring.profiles.active=prod"
```

在多环境部署中，通常 resources 目录下可能没有目标环境的配置文件，这主要是为了安全

考虑，开发环境不应该有线上环境的各种配置信息。可以将配置文件放到特定的目录中，并用 spring.config.location 指定配置文件的目录：

```
>java -jar -Dspring.config.location=file:env/ -Dspring.profiles.active=
test  target/ch11.deploy-0.0.1-SNAPSHOT.jar
```

配置文件位于当前目录的 env 目录下，profile 是 test，因此读取的是 application-test.properties 配置文件。

> **注意**：无论用上面哪种多环境配置方法，总是会覆盖已有的 application.properties。
>
> Spring Boot 如何找到配置文件：
>
> Spring Boot 应用默认读取了 application.properties 文件，实际上，Spring Boot 会自动搜索 classpath:、classpath:/config、file:、file:config/ 这些目录下的配置文件，优先级由低到高，file:config/的优先级最高。
>
> 这是系统属性 spring.config.location 默认的配置。spring.config.name 表示配置文件的名称，默认是 application。

## 8.4 @Profile 注解

系统属性 spring.profiles.active 可以指定使用哪种配置文件，如果指定 test，则 application-test.properties 会覆盖默认的 application.properties。

@Profile 注解可以结合@Configuration 和@Component 使用，以决定配置类是否生效。

```
@Configuration
public class DataSourceConf {
  @Bean(name = "dataSource")
  @Profile("test")
  public DataSource testDatasource(Environment env) {
    HikariDataSource  test = getDataSource(env);
    test.setMaximumPoolSize(10);
    return test;
  }
  @Bean(name = "dataSource")
  @Profile("prod")
```

```
    public DataSource prodSource(Environment env) {
        HikariDataSource  prod = getDataSource(env);
        prod.setMaximumPoolSize(200);
        return prod;
    }

    private HikariDataSource getDataSource(Environment env){
        HikariDataSource ds = new HikariDataSource();
        // 省略初始化代码
        return ds;
    }
}
```

@Profile 注解可以支持使用多种 profile，也可以使用“!”来排除特定 profile。

- @Profile({“test”,"prod"})，测试环境和线上环境生效；
- @Profile({“test”,"!prod"})，测试环境和非线上环境生效。

# 第 9 章 Testing 单元测试

前面一章对 Spring Boot 项目做了介绍，为了帮助开发人员编写高品质的程序，提升代码质量，以及对代码重构的支持，单元测试都发挥了极大的作用，本章将讲一下 Spring Boot 单元测试。

## 9.1 JUnit 介绍

JUnit 是一个由 Java 语言编写的开源的回归测试（回归测试是指重复以前全部或部分的相同测试）框架，由 Erich Gamma 和 Kent Beck 创建，用于编写和运行可重复的测试，它是用于单元测试框架体系 xUnit 的一个实例。所谓单元测试也就是白盒测试。JUnit 是 Java 开发使用最为广泛的框架。该框架也得到了绝大多数 Java IDE 和其他工具（如 Maven）的集成支持。同时，JUnit 还有很多的第三方扩展和增强包可供使用。

### 9.1.1 JUnit 的相关概念

JUnit 的相关概念如下表所示。

| JUnit 的相关概念 | 解　释 |
|---|---|
| 测试 | 一个以@Test 注释的方法定义了一个测试，为了运行这个方法，JUnit 会创建一个包含类的新实例，然后再调用这个被注释的方法 |
| 测试类 | 一个测试类是@Test 方法的容器 |

续表

| JUnit 概念 | 解　释 |
|---|---|
| Assert | 定义想测试的条件，当条件成立时，assert 方法保持沉默，条件不成立时则抛出异常 |
| Suite | Suite 允许你将测试类归成一组 |
| Runner | Runner 类用来运行测试，JUnit4 是向后兼容的，可以运行 JUnit3 的测试 |

## 9.1.2 JUnit 测试

JUnit 3.x 版本通过对测试方法的命名（test+方法名）来确定是否是测试，且所有的测试类必须继承 TestCase。JUnit 4.x 版本全面引入了注解来执行我们编写的测试，JUnit 中有两个重要的类（Assume 和 Assert），以及其他一些重要的注解（BeforeClass、AfterClass、After、Before、Test 和 Ignore）。其中，BeforeClass 和 AfterClass 在每个类加载的开始和结束时运行，需要设置 static 方法；而 Before 和 After 则在每个测试方法开始之前和结束之后运行。

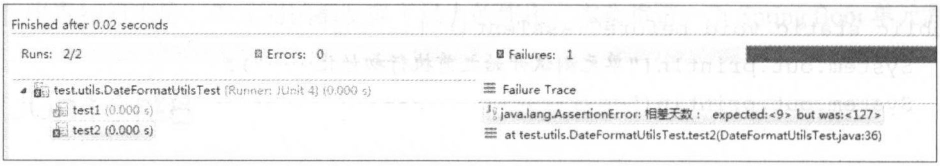
代码片段如下：

```
@BeforeClass
public static void beforeClassTest() {
    System.out.println("单元测试开始之前执行初始化……");
    System.out.println("------------------------------");
}
@Before
public void beforeTest() {
    System.out.println("单元测试方法开始之前执行……");
}
@Test
public void test1() {
     Date sd = DateFormatUtils.formatStr2Date("2017-05-16");
    Date ed = DateFormatUtils.formatStr2Date("2017-05-25");
    System.out.println("相差天数： " + DateFormatUtils.getBetweenDays(sd,ed));
    assertEquals("相差天数： ",9,DateFormatUtils.getBetweenDays(sd,ed));
}
@Test
public void test2() {
     Date sd = DateFormatUtils.formatStr2Date("2017-05-26");
    Date ed = DateFormatUtils.formatStr2Date("2017-09-30");
```

```
        System.out.println("相差天数： " + DateFormatUtils.getBetweenDays(sd,ed));
        assertEquals("相差天数： ",9,DateFormatUtils.getBetweenDays(sd,ed));
    }
    @After
    public void afterTest() {
        System.out.println("单元测试方法结束后执行……");
    }
    @AfterClass
    public static void afterClassTest() {
        System.out.println("------------------------------");
        System.out.println("单元测试结束之后执行……");
    }
```

JUnit 在执行每个@Test 方法之前，会为测试类创建一个新的实例。这有助于提供测试方法之间的独立性，并且避免在测试代码中产生意外的副作用。因为每个测试方法都运行于一个新的测试类实例上，所以不能在测试方法之间重用各个实例的变量值。

以上代码 JUnit 执行结果如下图所示，可以看出 test2 未通过测试，因为我们输入的预期值为 9 天，而实际上应该是 127 天，从错误结果的提示中也可以看出。




根据 Console 控制台输出，印证了我们上面所说的注解执行顺序：

```
单元测试开始之前执行初始化……
------------------------------
单元测试方法开始之前执行……
相差天数： 9
单元测试方法结束后执行……
单元测试方法开始之前执行……
相差天数： 127
单元测试方法结束后执行……
------------------------------
单元测试开始之前执行初始化……
```

有些人会问，JUnit 没有 main()方法作为入口是怎么运行的呢？其实在 org.junit.runner 包下

有个 JUnitCore.class，其中就有一个是标准的 main 方法，这就是 JUnit 入口函数。如此看来，它其实和我们直接在自己的 main 方法中跑我们要测试的方法在本质上是一样的。

### 9.1.3 Assert

为了进行测试验证，我们使用了 JUnit 的 Assert 类提供的 assert 方法。正如之前在实例中所看到的那样，我们在测试类中静态地导入了这些方法。另外，根据我们对静态导入的喜好，还可以导入 JUnit 的 Assert 类本身。下面列出了一些常用的 assert 方法：

- assertEquals("message",A,B)，判断 A 对象和 B 对象是否相等，这个判断在比较两个对象时调用了 equals()方法。
- assertSame("message",A,B)，判断 A 对象与 B 对象是否相同，之前的 assert 方法是检查 A 与 B 是否有相同的值（使用了 equals 方法），而 assertSame 方法则是检查 A 与 B 是不是同一个对象（使用的是 == 操作符）。
- assertTrue("message",A)，判断 A 条件是否为真。
- assertFalse("message",A)，判断 A 条件是否不为真。
- assertNotNull("message",A)，判断 A 对象是否不为 null。
- assertArrayEquals("message",A,B)，判断 A 数组与 B 数组是否相等。

### 9.1.4 Suite

JUnit 设计 Suite 的目的是一次性运行一个或多个测试用例，Suite 是一个容器，用来把几个测试类归在一起，并把它们作为一个集合来运行，测试运行器会启动 Suite，而运行哪些测试类由 Suite 决定。

示例如下：

```
@RunWith(Suite.class)
@SuiteClasses({TestSuite1.class, TestSuite2.class})
public class TestSuitMain{
  // 虽然这个类是空的，但依然可以运行 JUnit 测试，运行时，它会将 TestSuite1.class 和
  // TestSuite2.class 中的所有测试用例都执行一遍
}
```

## 9.2 Spring Boot 单元测试

Spring Boot 提供了一些实用程序和注解，用来帮助我们测试应用程序。测试由两个模块支持——spring-boot-test 包含了核心项目，而 spring-boot-test-autoconfigure 支持自动配置测试。

大多数开发者会使用 spring-boot-starter-test 导入 Spring Boot 测试模块，以及 JUnit、assertj、hamcrest 和其他一些有用的库。集成只需要在 pom 中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

### 9.2.1 测试范围依赖

如果使用 spring-boot-starter-test ‘Starter’ 来测试，就会发现提供的以下测试库：

- JUnit，标准的单元测试 Java 应用程序。
- Spring Test & Spring Boot Test，对 Spring Boot 应用程序的单元测试。
- Mockito，Java mocking 框架，用于模拟任何 Spring 管理的 Bean，比如在单元测试中模拟一个第三方系统 Service 接口返回的数据，而不会去真正调用第三方系统。
- AssertJ，一个流畅的 assertion 库，同时也提供了更多的期望值与测试返回值的比较方式。
- Hamcrest，库的匹配对象（也称为约束或谓词）。
- JSONassert，对 JSON 对象或者 JSON 字符串断言的库。
- JsonPath，提供类似 XPath 那样的符号来获取 JSON 数据片段。

这些通常是我们在编写测试时普遍用到的库。当然，如果这些库不能满足你的需要，也可以添加一些自己需要的额外的依赖库。本章会涉及 JUnit、Spring Boot Test、Mockito、JsonPath 工具。

### 9.2.2 Spring Boot 测试脚手架

Spring Boot 使用一系列注解来增强单元测试以支持 Spring Boot 测试。通常 Spring Boot 单元测试有类似如下的样子：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserServiceTest {
    @Autowired
    UserService userService; // 要测试的 service

    @Test
    public void testService(){

    }
}
```

@RunWith 是 JUnit 标准的一个注解，用来告诉 JUnit 单元测试框架不要使用内置的方式进行单元测试，而应使用 RunWith 指明的类来提供单元测试，所有的 Spring 单元测试总是使用 SpringRunner.class。

@SpringBootTest 用于 Spring Boot 应用测试，它默认会根据包名逐级往上找，一直找到 Spring Boot 主程序，也就是通过类注解是否包含@SpringBootApplication 来判断是否是主程序，并在单元测试的时候启动该类来创建 Spring 上下文环境。

> **注意：** Spring 单元测试并不会在每个单元测试方法前都启动一个全新的 Spring 上下文，因为这样太耗时。Spring 单元测试会缓存上下文环境，以提供给每个单元测试方法。如果你的单元测试方法改变了上下文，比如更改了 Bean 定义，你需要在此单元测试方法上加上@DirtiesContext 以提示 Spring 重新加载 Spring 上下文。

## 9.2.3 测试 Service

单元测试 Service 代码跟我们平常通过 Controller 调用 Service 代码来进行测试相比，有三个需要特别考虑的地方：

- 单元测试需要保证可重复的测试，因此希望 Service 测试完毕后，数据能自动回滚。
- 单元测试是开发过程中的一种测试手段，Service 依赖的其他 Service 还未开发完毕的情况下如何模拟呢？
- 大多数 Spring Boot 应用都是面向数据库的应用，如何在单元测试前模拟好要测试的场景呢？

对于第一个问题，Spring Boot 单元测试默认会在单元测试方法运行结束后进行事务回滚。

对于第二个问题，想必你也想到了，Spring Boot 会集成 Mockito 来模拟未完成的 Service

类（或者是在单元测试中不能随便调用的第三方接口 Service）。

对于第三个问题，Spring 引入了@Sql，在测试前执行一系列的 SQL 脚本来初始化数据。我们将用专门的一节来说明@Sql。

以下是一个 Service 单元测试的脚手架：

```
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
@Transactional
public class ServiceTest {
  @Autowired
  UserService userService;

  @MockBean
  private CreditSystemService creditSystemService;

  @Test
  public void testService() {
    int userId = 10;
    int expectedCredit = 100;
    given(this.creditSystemService.getUserCredit(anyInt())).willReturn
(expectedCredit);
    int credit = userService.getCredit(10);
    assertEquals(expectedCredit, credit);
  }
}
```

在这个例子中，我们要测试调用 UserService 的 getCredit 以获取用户积分。UserService 依赖 CreditSystemService 的 getUserCredit，通过 REST 接口从积分系统中获取用户的积分。UserService 定义如下：

```
@Service
@Transactional
public class UserServiceImpl implements UserService {

  @Autowired
  CreditSystemService creditSystemService;
```

```
    @Autowired
    UserDao userDao;

    @Override
    public int getCredit(int userId) {
        User user = userDao.single(userId);
        if(user!=null){
            return creditSystemService.getUserCredit(userId);
        }else{
            return -1;
        }

    }
}
```

因为单元测试不能实际调用 creditSystemService（假设会调用一个第三方系统），因此，我们在单元测试类中使用了@MockBean：

```
@MockBean
private CreditSystemService creditSystemService;
```

注解@MockBean 可以自动注入 Spring 管理的 Service，用来提供模式实现，因此 creditSystemService 变量在这里实际上并不是 CreditSystemServiceImpl 实例，而是一个通过 Mockito 工夹具创建的 CreditSystemService$MockitoMock$xxxxxx 实例（这里的 xxxxxx 是一组随机数字）。因此，在 Spring 上下文中，creditSystemService 实现已经被模拟实现代替了。

以下代码模拟了 Bean 的 getUserCredit 方法，无论传入什么参数，总是返回 100 积分：

```
given(this.creditSystemService.getUserCredit(anyInt())).willReturn(expec
tedCredit);
```

given 是 Mockito 的一个静态方法，用来模拟一个 Service 方法调用返回，anyInt()指示了可以传入任何参数，willReturn 方法说明这个调用将返回 100。

> Mockito 的用法将在 9.3 节详细说明。默认情况下，单元测试完毕，事务总是回滚，有时需要通过数据库查看数据测试结果而不希望事务回滚，可以在方法上使用@Rollback(true)。

## 9.2.4 测试 MVC

Spring Boot 可以单独测试 Controller 代码，用来验证与 Controller 相关的 URL 路径映射、文件上传、参数绑定、参数校验等特性。可以通过@WebMvcTest 来完成 MVC 单元测试，脚手架如下所示。

```
@RunWith(SpringRunner.class)
// 需要测试的 Controller
@WebMvcTest(UserController.class)
public class UserControllerTest {
  @Autowired
  private MockMvc mvc;

  @MockBean
  UserService userService;

  @Test
  public void testMvc() throws Exception {
    int userId = 10;
    int expectedCredit = 100;
    // 模拟 userService
    given(this.userService.getCredit(userId)).willReturn(100);
    // MVC 调用
    mvc.perform(get("/user/{id}",userId))
      .andExpect(content().string(String.valueOf(expectedCredit)));
  }
}
```

- @WebMvcTest 表示这是一个 MVC 测试，其参数可以传入多个待测试的 Controller 类，这里要测试的类是 UserController;
- MockMvc 是 Spring 提供的专门用于测试 Spring MVC 类。
- @MockBean 用来模拟实现，因为在 Spring MVC 测试中，Spring 容器并不会初始化@Service 注解的类，因此我们需要模拟 UserController 调用的所有 Service，这里就是 UserService;
- perform 完成一次 MVC 调用，Spring MVC Test 是 Servlet 容器内的一种模拟测试，实际上并不会发起一次真正的 HTTP 调用。

- get 方法模拟了一次 Get 请求，请求地址是/user/{id}，这里的${id}会被其后的参数 userId 代替，因此请求地址是/user/10；
- andExpect 表示请求期望的返回结果，比如返回的内容或者 HTTP 响应头等。

> 在 Spring MVC Test 中，带有@Service、@Component 的类不会自动被扫描注册为 Spring 容器管理的 Bean。
>
> MockMvc 用来在 Serlvet 容器内对 Controller 进行单元测试，并非发起了 HTTP 请求调用 Controller。

## 9.2.5 完成 MVC 请求模拟

MockMvc 的核心方法如下：

```
public ResultActions perform(RequestBuilder requestBuilder)
```

RequestBuilder 类可以通过使用 MockMvcRequestBuilders 的 get、post、multipart 等方法来实现，以下是一些常用的例子。

模拟一个 Get 请求：

```
mockMvc.perform(get("/hotels?foo={foo}", "bar"));
```

模拟一个 Post 请求：

```
mockMvc.perform(post("/hotels/{id}", 42);
```

模拟文件上传：

```
mockMvc.perform(multipart("/doc").file("file", "文件内容".getBytes("UTF-8")));
```

模拟请求参数：

```
// 模拟提交 message 参数
mvc.perform(get("/user/{id}/{name}", userId,name).param("message", "hello"));
// 模拟一个 checkbox 提交
mvc.perform(get("/user/{id}/{name}", userId,name).param("job", "IT","gov")
    .param(...));
```

```
// 直接使用 MultiValueMap 构造参数
LinkedMultiValueMap params  = new  LinkedMultiValueMap();
params.put("message","hello");
params.put("job","IT");
params.put("job","gov");
mvc.perform(get("/user/{id}/{name}", userId,name).param(params));
```

模拟 Session 和 Cookie：

```
mvc.perform(get("/user.html").sessionAttr(name, value));
mvc.perform(get("/user.html").cookie(new Cookie(name,value)));
```

设置 HTTP Body 内容，比如提交的 JSON：

```
String json = ....;
mvc.perform(get("/user.html").content(json));
```

设置 HTTP Header：

```
mvc.perform(get("/user/{id}/{name}", userId,name)
  .contentType("application/x-www-form-urlencoded") // HTTP 提交内容
  .accept("application/json") // 期望返回内容
  .header(header1,value1))// 设置 HTTP 头
```

## 9.2.6 比较 MVC 的返回结果

从 9.2.5 节知道，perform 方法返回 ResultActions 实例，这个类代表了 MVC 调用的结果。它提供一系列 andExpect 方法来对 MVC 调用结果进行比较，如：

```
mockMvc.perform(get("/user/1"))
  .andExpect(status().isOk()) // 期望成功调用，即 HTTP Status 为 200
  // 期望返回内容是 application/json
  .andExpect(content().contentType(MediaType.APPLICATION_JSON))
  .andExpect(jsonPath("$.name").value("Jason"));// 检查返回内容
```

也可以对 Controller 返回的 ModeAndView 进行校验，如比较返回的视图：

```
mockMvc.perform(post("/form"))
```

```
    .andExpect(view().name("/success.btl"));
```

比较 Model：

```
mockMvc.perform(post("/form"))
    .andExpect(status().isOk()
    .andExpect(model().size(1))
    .andExpect(model().attributeExists("person")
    .andExpect(model().attribute("person","xiandadfu"));
```

比较 forward 或者 redirect：

```
mockMvc.perform(post("/login"))
    .andExpect(forwardedUrl("/index.html"));// 或者
redirectedUrl("/index.html")
```

比较返回内容，使用 content()：

```
andExpect(content().string("hello world"));
// 返回内容是 XML，并且与 xmlCotent 一样
andExpect(content().xml(xmlCotent));
// 返回内容是 JSON，并且与 jsonContent 一样
andExpect(content().json(jsonContent));
andExpect(content().bytes(bytes);
```

XML 方法和 JSON 方法用来比较返回值和期望值的相似程度，比如返回值是{"success":true}，期望值是{"success" : true}，两者依然匹配。

### 9.2.7 JSON 比较

Spring Boot 内置了 JsonPath 来比较返回的 JSON 内容，通常类似如下代码：

```
String path = "$.success";
mvc.perform(get("/user/{id}/{name}",
userId,name)).andExpect(jsonPath(path).value(true));
```

这段代码期望返回的 JSON 的 success 属性是 true，$代表了 JSON 的根节点。

以下是一个 JSON 文档，我们可以用 JsonPath 表达式来抽取 JSON 节点的内容：

```
{
  "store": {
    "book": [
      {
        "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      {
        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      }
    ]
  }
}
```

下表列举了一些 Spring Boot 常用的 JsonPath 场景，更多的 JsonPath 使用方法请参考官网 https://github.com/json-path/JsonPath。

| Path | 返 回 值 |
|---|---|
| $.store.book[*].author | 所有 book 的 author |
| $..author | 所有 author |
| $.store.* | store 下的所有节点 |
| $..book[2] | 第三个 book 节点 |
| $..book.length() | 返回 book 的个数 |
| $..book[0,1] | 第一个和第二个 book 节点 |
| $.store.book[?(@.price < 10)] | 所有 price 小于 10 的节点，[?()] 包含了表达式，@指示的是当前节点 |

## 9.3 Mockito

在测试过程中，对那些不容易构建的对象用一个虚拟对象来代替测试的方法称为 Mock 测试。

目前，在 Java 阵营中主要的 Mock 测试工具有 Mockito、JMock、EasyMock 等，Spring Boot 内置了 Mockito，本节将详细介绍 Mockito 的用法。

Mockito 可以模拟任何类和接口，模拟方法调用的返回值，模拟抛出异常等。Mockito 实际上同时也记录调用这些模拟方法的输入/输出和顺序，从而可以校验这些模拟对象是否被正确的顺序调用，以及按照期望的属性被调用。

本节考虑一个积分系统未完成的情况下，模拟 CreditSystemService：

```
public interface CreditSystemService {
  public int getUserCredit(int userId);
  public boolean addCedit(int userId,int score);
}
```

## 9.3.1 模拟对象

为了学习 Mockito，将暂时脱离 Spring Boot 容器测试，这样节省启动时间。单元测试使用 MockitoJUnitRunner 来运行单元测试，以下代码是一个学习 Mockito 的脚手架代码：

```
import static org.mockito.Mockito.*;

@RunWith(MockitoJUnitRunner.class)
public class CreditServiceMockTest {
    @Test
    public void test(){
        int userId = 10;
        // 创建 Mock 对象
        CreditSystemService creditService= mock(CreditSystemService.class);
        // 模拟 Mock 对象调用，传入任何 int 值都将返回 100 积分
        when(creditService.getUserCredit(anyInt())).thenReturn(1000);

        // 实际调用
        int ret = creditService.getUserCredit(10);
        // 比较期望值和返回值
        assertEquals(1000,ret);
    }
}
```

org.mockito.Mockito 包含了一系列我们会用到的模拟测试方法，如 mock、when、thenReturn 等。

通过 mock 方法可以模拟任何一个类或者接口，比如模拟一个 java.util.List 接口实现，或者模拟一个 java.util.LinkedList 类实现：

```
LinkedList mockedList = mock(LinkedList.class);
List list = mock(List.class);
```

## 9.3.2 模拟方法参数

Mockito 提供 any 方法模拟方法的任何参数，比如：

```
when(creditService.getUserCredit(anyInt())).thenReturn(1000);
```

anyInt 指的是无论传入任何参数，总是返回 100，也可以使用 any 方法：

```
when(creditService.getUserCredit(any(int.class))).thenReturn(1000);
```

单元测试中，大多数时候不推荐使用 any 方法，因为为模拟的对象提供更明确的输入/输出才能更好地完成单元测试，比如在 9.2 节 Spring Boot 单元测试中，通过 UserService 调用 CreditSystemService 来获取用户积分，传入的参数 userId 是明确的，因此我们最好用具体的方法参数来代替 any。

```
int userID= 10;
// 模拟某个场景需要调用 creditService 的 getUserCredit
when(creditService.getUserCredit(eq(userId)).thenReturn(1000);
```

以上这一段代码模拟了当传入参数是 10 的时候，返回 100 积分。因此，如果在单元测试中，被测试代码并未按照预期的参数传入的时候，Mockito 会报错。

```
int ret = creditService.getUserCredit(11);
```

这段代码并未按照预期传入 10，而是传入了 11，运行单元测试会报出如下错误信息，提示这一行代码参数传入是否正确：

```
[MockitoHint] CreditServiceMockTest.test (see javadoc for MockitoHint):
[MockitoHint] 1. Unused... -> at
```

```
com.bee.sample.ch9.test.mock.CreditServiceMockTest.test(CreditServiceMockTes
t.java:20)
    [MockitoHint] ...args ok? -> at
com.bee.sample.ch9.test.mock.CreditServiceMockTest.test(CreditServiceMockTes
t.java:23)
```

Mockito 不仅仅能模拟参数的调用和返回值，而且也记录了模拟对象是如何调用的，因此，如果我们模拟的调用并未被实际调用，Mockito 也会报错，指示某些模拟并未使用。我们可以通过 verify 方法来更为精确地校验模拟对象是否被调用。

比如某些业务场景，我们假设肯定会调用两次 getUserCredit 方法。如果没有调用两次，则判断单元测试失败。

```
// 模拟 getUserCredit
when(creditService.getUserCredit(eq(userId))).thenReturn(1000);
// 实际调用，模拟一个业务场景会调用两次 getUserCredit 接口
int ret = creditService.getUserCredit(userId);
ret = creditService.getUserCredit(userId);
// 比较期望值和返回值
assertEquals(1000,ret);
verify(creditService,times(2)).getUserCredit(eq(userId));
```

verify 方法包含了模拟的对象和期望的调用次数，使用 times 来构造期望调用的次数，如果在业务调用中只发生了一次 getUserCredit 调用，那么在 Mockito 在单元测试中会抛出如下异常信息：

```
    org.mockito.exceptions.verification.TooLittleActualInvocations:
    creditSystemService.getUserCredit(10);
    Wanted 2 times:
    -> at com.bee.sample.ch9.test.mock.CreditServiceMockTest.test
(CreditServiceMockTest.java:29)
    But was 1 times:
    -> at com.bee.sample.ch9.test.mock.CreditServiceMockTest.test
(CreditServiceMockTest.java:26)
        at com.bee.sample.ch9.test.mock.CreditServiceMockTest.test
(CreditServiceMockTest.java:29)
```

这段错误提示指示了期望有两次调用，但实际上只发生了一次调用。

因为 Mockito 能记录模拟对象的调用，因此除了模拟调用对象方法的次数，还能验证调用的顺序。使用 inOrder 方法：

```
// 创建 Mock 对象
CreditSystemService creditService = mock(CreditSystemService.class);
// 模拟 Mock 对象调用
when(creditService.getUserCredit(eq(userId))).thenReturn(1000);
when(creditService.addCedit(eq(userId), anyInt())).thenReturn(true);

// 实际调用，先获取用户积分，然后增加 10 分
int ret = creditService.getUserCredit(userId);
creditService.addCedit(userId, ret + 10);

// 验证调用顺序，确保模拟对象先被调用 getUserCredit，然后再被调用 addCedit 方法
InOrder inOrder = inOrder(creditService);
inOrder.verify(creditService).getUserCredit(userId);
inOrder.verify(creditService).addCedit(userId, ret + 10);
```

> 建议在学习 Mockito 的过程中，可以故意制造一些错误的调用参数和调用顺序，以了解 Mockito 如何展示报错信息，这样有助于深入掌握 Mockito。故意制造错误是学习开发的窍门。

## 9.3.3 模拟方法返回值

当使用 when 来模拟方法调用的时候，可以使用 thenReturn 来模拟返回的结果：

```
when(creditService.getUserCredit(eq(userId))).thenReturn(1000);
when(creditService.addCedit(eq(userId), anyInt())).thenReturn(true);
```

也可以使用 thenThrow 来模拟抛出一个异常，比如：

```
CreditSystemService creditService = mock(CreditSystemService.class);
// 模拟 Mock 对象调用
when(creditService.getUserCredit(eq(userId))).thenReturn(1000);
when(creditService.getUserCredit(lt(0))).thenThrow(new
IllegalArgumentException("userId 不能小于 0"));
// 实际调用
int ret = creditService.getUserCredit(-1);
```

eq 表示参数相等的情况下，lt 表示参数小于 0 的情况下，将抛出异常。

有些情况下，模拟的方法并没有返回值，可以使用 doThrow 方法来抛出异常：

```
// 模拟 List 对象
List list = mock(List.class);
doThrow(new UnsupportedOperationException("不支持 clear 方法调用")).
when(list).clear();
// 实际调用，将抛出异常
list.clear();
```

这一段代码模拟了一个 List 对象的 clear 方法调用将抛出异常。

## 9.4 面向数据库应用的单元测试

对于绝大部分 Spring Boot 应用来说，都包含了数据库 CRUD 操作。在单元测试中，我们可以通过模拟 Dao 类来返回预期的 CRUD 结果。但对于复杂的面向数据库应用，我们有时候不但需要比较业务调用结果是否与期望的一致，我们更期望业务调用完毕，数据库各个表的行和列与期望的数据库行和列的值是一样的。

比如一个更新用户手机号码的业务调用，我们期望调用完毕后，数据库的 User 表中这个用户的 mobile 字段与我们期望的 mobile 一致。

比如在工作流引擎中，当调用引擎的流程结束的时候，我们期望 Workflow 表中这条流程的 status 状态是结束状态。

面向数据库应用的单元测试是绝大多数 Java Web 框架回避的一个话题，本章将结合 Spring Boot 和我的个人经验来说明如何进行面向数据库应用的单元测试。

### 9.4.1 @Sql

Spring Boot 提供了@Sql 来初始化数据库。需要为单元测试准备一个新的数据库，这个新的数据库通常不包含任何数据，或者只包含一些必要的字典类型的数据和初始化数据。

注解@Sql 可以引入一系列 SQL 脚本来进一步模拟测试前的数据库数据，以下是单元测试脚手架：

```
@RunWith(SpringRunner.class)
@SpringBootTest
```

```
@ActiveProfiles("test")
@Transactional
public class UserDbTest {

  @Autowired
  UserService userService;

  @Test
  @Sql({"user.sql"}) // 初始化一条主键为 1 的用户数据
  public void upateNameTest(){

     User user = new User();
     user.setId(1);
     user.setName("hello123");
     // 修改用户名称
     boolean success = userService.updateUser(user);
     assertTrue(success);
  }
}
```

这段代码与之前我们看到的 Spring Boot 单元测试脚手架差不多，有两个区别：

- @ActiveProfiles("test")，因为我们需要连接一个专门用于单元测试的数据库，因此我们激活了 test 作为 profile。我们可以创建一个新的名为 application-test.property 的 Spring Boot 配置文件，进行单元测试的时候将读取此配置文件。关于 Profile，可以参考第 8 章多环境部署。application-test.property 的内容如下：

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/orm-test?useUnicode=true&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

orm-test 数据库与 ORM 类似，不同的是 orm-test 不包含任何数据，仅仅用于单元测试。

- @Sql 注解可以包含多个 SQL 脚本，用来在单元测试方法前初始化数据。如果 SQL 脚本没有以"/"开头，则默认在测试类所在包下。否则，则从根目录搜索。也可以使用 classpath:、file:、http:作为前缀的资源文件。上述例子中的@Sql({"user.sql"})相当于@Sql({"classpath:com/bee/sample/ch9/test/db/user.sql"})，user.sql 的内容如下：

```
INSERT INTO `user` (id,`name`, `department_id`) VALUES (1,'lijz', '1');
```

使用@Sql 尽管能在单元测试前初始化所需要的数据，但要比较调用业务方法后的数据库中的数据是否与我们期望的数据库数据是否一致，Spring 现在还没有提供更多直接的办法，我们只能通过 Dao 从数据库加载数据来进行比较。以 BeetlSQL 为例，可以做如下改进：

```
@Autowired
UserDao userDao;

@Test
@Sql({"user.sql"})
public void upateNameTest(){
  User user = new User();
  user.setId(1);
  user.setName("hello123");
  boolean success = userService.updateUser(user);
  User dbUser = userDao.unique(1);
  assertEquals(dbUser.getName(),"hello123");
}
```

在单元测试中注入 BeetlSQL 的 UserDao，并在单元测试中调用 unique 方法以获得 User 实例来进行比较。

### 9.4.2 XLSUnit

在前面一节中，Spring 提供了@Sql 来初始化单元测试数据库，但有三个问题：

- 通过脚本来初始化单元测试数据库使得初始化数据不够直观，特别是需要初始化的数据库涉及多张表、多条数据的时候。
- 初始化 SQL 脚本中的数据不得不与单元测试中的代码数据写死，比如 user.sql 模拟了一条主键为 1 的 user 数据，单元测试中也必须用 1 来作为 User 对象的 id 属性。
- 单元测试到底能覆盖多少业务场景，对于项目经理或者需求人员并不明显，需要有一种直观的表达方式。

除了这三个问题，在业务方法调用完毕，需要比较数据库的数据与期望的数据是否一致的时候，还需要调用 Dao 加载数据后进行一一比较，非常不直观。以用户买房贷款为例子，一笔

贷款，可能在数据库中生成数十行的还款计划记录。逐一比较生成的记录是否与期望一致，这需要编写许多比较代码。

DBUnit 是另外一款单元测试工具，在一定程度上弥补了 Spring 面向数据库单元测试的不足。通过 XML 来模拟和比较数据，但是模拟和比较数据都非常不直观，既然是面向数据库单元测试，为什么不能用 XLS 初始化要测试前的数据，并且还用 XLS 来比较单元测试后的期望数据呢？

XLSUnit 是笔者所在公司开发的一个面向数据库单元测试工具，它通过 Excel 的工作表模拟数据库初始化数据，用其他工作表来模拟单元测试后的期望结果，只需要写少量单元测试代码，维护直观的 Excel 文件，就可以完成单元测试。

XLSUnit 可以加载 Excel 文件，从而初始化单元测试数据库。单元测试方法可以从 XLSUnit 中获取 Excel 中的数据作为参数来完成业务调用。调用完毕后，可以调用 XLSUnit 方法来比较数据库现有数据与 Excel 文件某个工作表描述的数据是否一致。XLSUnit 简化了面向数据库应用的单元测试代码，使得单元测试更容易维护和观察。

## 9.4.3 XLSUnit 的基本用法

可以拥有任意多个 Excel 文件，每个文件从概念上讲都可以包含一个初始化数据的多种测试场景，Excel 文件由三部分组成：

- 第一个是工作表，这个工作表用来介绍当前 Excel 文件所测试的业务，一般包含一个目录，快捷指向每个场景测试。
- 第二个表是数据库初始化数据，可以包含多个表的初始化数据，也可以在初始化数据定义前，定义一些变量，这些变量可以用在数据库初始化数据中，也可以用在 JUnit 代码中。
- 第三个表后的每个工作表都是场景测试表，对应了单元测试完毕后，期望的数据库的样子。通常用红色背景标注在表的列名上，这样表示只让 XLSUnit 比较关心的列。同时，也能提醒阅读者编写期望值。

### 9.4.3.1 编写目录表

目录表示 Excel 的第一个工作区，可以是任意内容。比如可以介绍一下这个 Excel 所代表的单元测试内容，单元测试代码的类名可以通过 Excel 来找到单元测试。

| com.coamc.workflow.core.EngineTest | 描　述 |
| --- | --- |
| 启动流程测试 | 用于启动流程测试，验证接口 startWorklfow |

续表

| com.coamc.workflow.core.EngineTest | 描　述 |
|---|---|
| 获取代办任务测试 | 验证用户获得代办任务，验证接口 getTodoList |
| 审批通过“业务审核” | 第一个环节审批通过 |
| 拒绝“业务审核” | 第一个环节打回重填 |
| 取消“业务审核” | 第一个环节取消流程，流程自动结束 |

以上目录表在第一个 Excel 工作表中，内容可以是任意格式，XLSUnit 并不处理第一个工作表，内容通常包含了对单元测试的描述。第一列是目录，建立超链接，链接到每个场景测试。第二列则是对每个单元测试的描述。最顶行标注了这个 Excel 文件被哪个单元测试类用到。

9.4.3.2　编写输入表

以注释符号“##:”开头的表示是一个注释语句，用在 Excel 的行首：

```
##: 这是定义一个工作流类型
```

符号“##”+小写变量名定义了一组变量，这组变量可以在随后模拟数据库数据的时候使用，也可以在 Java 代码里复用，这解决了 Spring 测试注解@sql 的问题，单元格的类型决定了数据类型，尤其要注意字符和数字的区别。

| | | |
|---|---|---|
| ##workflow | | |
| code | name | desc |
| 1199 | 任务审批工作流 | ${workflow.code+"-"+workflow.name} |
| | | |
| ##user | | |
| id | name | orgId |
| 1 | 流程发起人 | 1 |

以上声明了 workflow 变量，包含 workflow 变量和 user 变量，workflow 包含属性 code、name 和 desc。其中 desc 使用了 Beetl 表达式，user 包含了 id、name。

正如 Java 中使用“{}”来分块，XLSUnit 使用空行来分割每一段内容，包括注释、变量定义、数据库表。

XLSUnit 在 Excel 中声明的变量可以在 Java 代码中引用，也可以在随后的模拟数据库表中使用。

符号“##”+大写变量名定义了一个表格数据，其下一行是列名字，剩下所有行都是数据库数据。

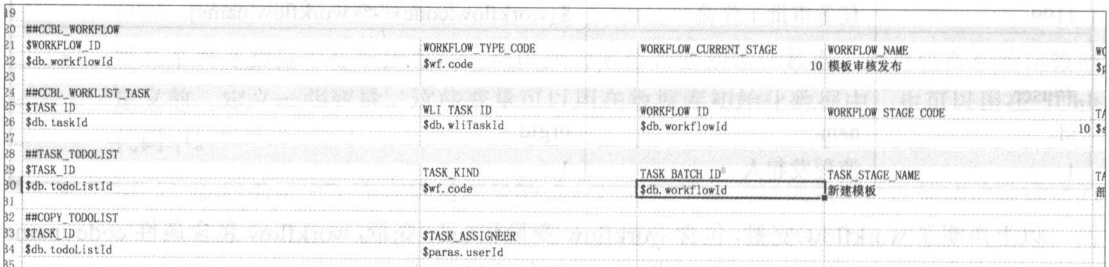
| ##WORKFLOW_TYPE | | | | |
|---|---|---|---|---|
| $ID | CODE | NAME | DESC | CREATE_TIME |
| $id | ${workflow.code} | ${workflow.name} | ${workflow.desc} | $fn.date |
| $id1 | …… | …… | …… | …… |

以上 Excel 片段模拟了工作流 WORKFLOW_TYPE 数据，定义了一个工作流的基本信息。基本信息沿用了定义的变量，有几个地方需要特别关注：

- 主键需要用$标示，以提醒这是数据库主键，如主键 ID 写成$ID。
- $id，因为字段 ID 是自增主键，所以我们用$xxx 表示这一行数据存入数据库后，获取自增主键并赋值给 id 变量。在初始化数据库的数据中，变量可以用$开头引用，或者使用${}表达式引用，如果 XLSUnit 发现的变量并未定义，则表示此变量需要从数据库取回并赋值，常用在自增或者数据库触发器执行后添加的数据。上面的例子中，当 WORKFLOW_TYPE 表的数据库数据初始化完毕，id 和 id1 自动被 XLSUnit 赋值为相应的自增值。
- $fn.date，表示获取当前日期，其他函数还有$fn.seq，生成一个 Long 类型的自增的序列。

### 9.4.3.3 编写场景测试工作表

以下是一个输入表，包含初始化 4 张表的测试：



| | | | | |
|---|---|---|---|---|
| 20 | ##CCBL_WORKFLOW | | | |
| 21 | $WORKFLOW_ID | WORKFLOW_TYPE_CODE | WORKFLOW_CURRENT_STAGE | WORKFLOW_NAME |
| 22 | $db.workflowId | $wf.code | 10 | 模板审核发布 |
| 24 | ##CCBL_WORKLIST_TASK | | | |
| 25 | $TASK_ID | WLI_TASK_ID | WORKFLOW_ID | WORKFLOW_STAGE_CODE |
| 26 | $db.taskId | $db.wliTaskId | $db.workflowId | 10 |
| 28 | ##TASK_TODOLIST | | | |
| 29 | $TASK_ID | TASK_KIND | TASK_BATCH_ID | TASK_STAGE_NAME |
| 30 | $db.todoListId | $wf.code | $db.workflowId | 新建模板 |
| 32 | ##COPY_TODOLIST | | | |
| 33 | $TASK_ID | $TASK_ASSIGNEER | | |
| 34 | $db.todoListId | $paras.userId | | |

目录 输入 提交多人处理任务

场景测试表代表调用测试方法后数据库期望的样子，场景测试表与输入表一样，主要不同的地方是需要用红色背景色标注在表的列字段上，用来指示 XLSUnit 仅仅比较需要关注的列。

例如流程的状态变化等，以下是“提交多人处理任务”期望的数据库表：

| | A | B | C | D | E | |
|---|---|---|---|---|---|---|
| 1 | ##CCBL_WORKFLOW | | | | | |
| 2 | $WORKFLOW_ID | WORKFLOW_TYPE_CODE | WORKFLOW_CURRENT_STAGE | WORKFLOW_NAME | WORKFLOW_CREATOR | CREATE_TIM |
| 3 | $db.workflowId | $wf.code | 11 | 模板审核发布 | $paras.userId | $fn.date |
| 4 | | | | | | |
| 5 | ##CCBL_WORKLIST_TASK | | | | | |
| 6 | $TASK_ID | WLI_TASK_ID | WORKFLOW_ID | WORKFLOW_STAGE_CODE | TASK_TYPE_ID | COMMENTS |
| 7 | $db.taskId | $db.wliTaskId | $db.workflowId | 10 | $stage.one | |
| 8 | $ret.taskId | $db2.wliTaskId | $db.workflowId | 11 | $stage.two | |
| 9 | | | | | | |
| 10 | ##TASK_TODOLIST | | | | | |
| 11 | $TASK_ID | TASK_KIND | TASK_BATCH_ID | TASK_STAGE_NAME | TASK_BATCH_NAME | TASK_SUBJE |
| 12 | $db.todoListId | $wf.code | $db.workflowId | 新建模板 | 部门审核 | 新建模板并 |
| 13 | $ret.todoListId | $wf.code | $db.workflowId | 部门审核 | 部门审核 | 新建模板并 |
| 14 | | | | | | |
| 15 | ##COPY_TODOLIST | TASK_ID=#ret.todoLis | Id# ORDER BY TASK_ASSIGNEER ASC | | | |
| 16 | $TASK_ID | $TASK_ASSIGNEER | | | | |
| 17 | $ret.todoListId | $rec.userId | | | | |
| 18 | $ret.todoListId | $rec2.userId | | | | |
| 19 | | | | | | |
| 20 | | | | | | |
| 21 | | | | | | |
| 22 | | | | | | |
| 23 | | | | | | |
| 24 | | | | | | |
| 25 | | | | | | |
| 26 | | | | | | |
| 27 | | | | | | |
| 28 | | | | | | |
| 29 | | | | | | |
| 30 | | | | | | |
| 31 | | | | | | |
| 32 | | | | | | |
| 33 | | | | | | |
| 34 | | | | | | |
| 35 | | | | | | |
| 36 | | | | | | |

目录 输入 提交多人处理任务 +

这个场景测试工作表包含了需要比较的 4 张表的数据变化，红色背景的列名是需要比较的数据。比如 CCBL_WORKFLOW 表的相应数据从 10 更改为 11，CCBL_WORKLIST_TASK 表增加了一条记录。

数据库与 XLSUnit 的场景比较有两种方式，一种是按照主键进行比较，CCBL_WORFLOW 代表某个工作流实例的状态，按照 $db.workflowId 比较，比较的列名是 WORKFLOW_CURRENT_ STAGE。

另外一种是提供 SQL 查询语句，将查询结果与场景进行比较，如 COPY_TODOLIST 表，因为“提交多人任务处理”可能会生成多条记录，因此我们提供了 SQL 语句来查询数据库值与期望的场景表中的值是否一致，这里的 SQL 语句使用 BeetlSQL。

#### 9.4.3.4 XLSUnit API

XLSUnit 的两个核心类：

- XLSParser，用于解析 Excel 表格，读取 Excel 定义的变量，存入 Excel 定义的数据到数据库，以及比较场景定义的数据与数据库数据是否一致。
- VariableTable，变量表，包含了 Excel 中定义的变量，可以通过 find 方法查找。

以下是一个测试工作流启动的单元测试例子，需要构造 XLSParser：

```
// XLSUnit 核心类，用来解析 Excel，对比 Excel
XLSParser workflowParser = null;
// 被测试的接口
@Autowired WorkflowService workflowService;
// 模拟工作流的用户系统接口，工作流通常采用第三方系统的用户和组织模型
@MockBean
private ThirdPatyUserService userService;

@Autowired
protected SQLManager sqlManager;

@Before
public void init() {
  super.init();
  // XLSUnit 的数据访问方式使用 BeetlSQL 实现
  DBAccess dbAccess = new BeetlSqlDBAccess(sqlManager);
  // Excel 文件根目录
  XLSFileLoader loader = new XLSFileLoader("...");
  // 构造核心类
  workflowParser = new XLSParser(loader, "测试一.xlsx", dbAccess,
                    new RowHolderFacotoy.RowBeetlSQLHolderFactory());
}
```

有了这个 WorkflowParser，就可以测试各种场景，以下是测试“提交多人任务处理”：

```
@Test
public void testWorkflow() {
  // 执行某个测试场景的初始化工作，初始工作流定义表
  VariableTable vars = new VariableTable();
  workflowParser.init(vars);

  // 开始测试，从场景中获取要测试的数据
  workflowParser.prepare("提交多人任务处理", vars);
  Integer type = (Integer) vars.find("workflow.type");
  Integer userId = (Integer) vars.find("user.id");
```

```
    // 模拟工作流引擎获取用户信息，workflowService 内部代码会调用第三方系统接口来获取
用户信息
    when(userService.getUserOrg(eq(userId)).thenReturn(vars.findInteger
("user.orgId"));

    // 开始测试，启动工作流
    workflowService.startWrokflow(type,userId);

    // 开始比较，数据库里的数据与场景"提交多人任务处理"的数据是否一致
    workflowParser.test("提交多人任务处理", vars);
  }
```

真正初始化数据库和比较测试结果的代码仅仅两行，可以看到 XLSUnit 能简化面向数据库的单元测试。

> 由于篇幅有限，用一节来完全讲清楚 XLSUnit 的用法还是比较困难的，可以访问 ibeetl.com 官网来学习 XLSUnit，以及下载更多例子。

# 第 10 章 REST

chapter 10

8 年前我在 HP 的时候，曾参与设计某央企消息系统，该系统需要连接各省业务系统，也要连接数百个全国业务系统，提供业务信息交换等功能。每日处理的消息多达数亿条，而每条消息包含的业务信息非常大。当时参与该项目竞标的都是国内外最好的消息中间件厂商。

在经过数月的测试和评估后，HP 的基于 HTTP 的 Web 应用系统获得了客户的赞赏。尽管在消息处理速度上和可靠性上，HTTP 方案会逊于消息中间件，但 HTTP 方案被认可，还是因为具有以下优点：

- HTTP 协议较为简单，协议公开透明。
- HTTP 的成熟性。HTTP 有大量可选的 Web 服务器，如 Tomcat、Undertow、Jetty，以及商业服务器，还有配套的 Web 框架，如本书讲的 Spring Boot，以及配套的负载均衡工具 Nginx、Apache，还有性能监控工具等，可伸缩性和性能都非常优秀。
- 技术的松耦合。Web 方案并没有与其他技术捆绑在一起，但又能将这些技术任意集成进来。
- Web 方案成本低。Web 方案，无论是购买商业服务器，还是选用技术开发人员，成本都相对较低，可选范围广。

HP 方案的成功并不是偶然的，而是当时“Web 即应用平台”已经深入人心，当系统在对外提供服务时，无论是系统之间，还是终端与系统之间（如 PC 终端、移动终端、平板终端等），双方总是优先会讨论建立在 Web 方式上的接口，如较早的 XML-RPC、曾经流行的 WebService，

以及现在广泛使用的 RESTful 风格接口。

本章将介绍 Restful 风格接口，并通过 Spring Boot 来实现 RESTful，最后会介绍 Swagger 工具，来增强 RESTful 的维护开发。

# 10.1 REST 简介

REST 这个词是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的，Fielding 是一个非常重要的人，他是 HTTP 协议（1.0 版和 1.1 版）的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席。所以，他的这篇论文发表后，就引起了广泛关注，并且对互联网开发产生了深远的影响。

Fielding 将他对互联网软件的架构原则定名为 REST，即 Representational State Transfer 的缩写，翻译为“表现层状态转化”。如果一个架构符合 REST 原则，就称它为 RESTful 架构。

要理解 RESTful 架构，最好的方法就是去理解 Representational State Transfer 这个词组到底是什么意思，它的每一个词代表了什么含义。如果你把这个名称搞懂了，也就不难体会 REST 是一种什么样的设计。

- 资源（Resources）

REST 的名称“表现层状态转化”中省略了主语。“表现层”其实指的是“资源”（Resources）的“表现层”。

所谓“资源”，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实体。你可以用一个 URI（统一资源定位符）指向它，每种资源对应一个特定的 URI。要获取这个资源，访问它的 URI 就可以，因此 URI 就成了每一个资源的地址或独一无二的识别符。

- 表现层（Representation）

“资源”是一种信息实体，它可以有多种外在表现形式。我们把“资源”具体呈现出来的形式称为它的“表现层”（Representation）。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式；图片可以用 JPG 格式表现，也可以用 PNG 格式表现。

URI 只代表资源的实体，不代表它的形式。严格地说，有些网址最后的“.html”后缀名是不必要的，因为这个后缀名表示格式，属于“表现层”范畴，而 URI 应该只代表“资源”的位置。它的具体表现形式应该在 HTTP 请求的头信息中用 Accept 和 Content-Type 字段指定，这两个字段才是对“表现层”的描述，不过在大部分应用中，通过后缀区分表现层已经足够了。

- 状态转化（State Transfer）

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及数据和状态的变化。

HTTP 协议是一个无状态互联网通信协议，这意味着所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。而这种转化是建立在表现层之上的，所以就是“表现层状态转化”。

客户端用到的手段只能是 HTTP 协议。具体来说，HTTP 协议里有 5 个常用的表示操作方式的动词：GET、POST、PUT、DELETE、PATCH。它们分别对应 5 种基本操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源，PATCH 用来更新资源的部分属性。

> URI（Uniform Resource Identifier）是统一资源标识符，如 http://baidu.com,ftp://xxxx。而 URL（Uniform Resource Locator）是统一资源定位符，是较早的概念，专门用于 HTTP 协议。
>
> 这一节 REST 定义参考了阮一峰对 REST 的理解和百度词条。

## 10.1.1 REST 风格的架构

现在流行的各种 Web 框架，包括 Spring Boot 都支持 REST 开发，REST 并非是一种技术或者规范，而是一种架构风格，这种架构风格逐渐被各种编程语言的 Web 框架所支持。它包括了 REST 架构中如何标识资源，如何标识操作接口及操作的版本，如何标识操作的结果等，主要内容如下：

- 使用“api”作为 Web 上下文；
- 增加版本标识；
- 标识资源；
- REST 中的 HTTP Method；
- REST 中的 HTTP Status。

## 10.1.2 使用“api”作为上下文

建议使用“api”作为上下文，如：

```
http://192.168.0.1/api
```

也有的使用“api”作为二级域名：

```
http://api.xxxx.com
```

### 10.1.3 增加一个版本标识

```
http://192.168.0.1/api/v1.1
```

也有的做法是将版本信息放到 HTTP 头中，但这里推荐还是通过 URL 来体现，这样使得 REST 的相关代码更加容易阅读。

### 10.1.4 标识资源

将资源名称放到 URL 中，如果资源有层级关系，则放入层级关系：

```
http://192.168.0.1/api/v1.1/user
```

如果用户属于系统管理，也可以这么写：

```
http://192.168.0.1/api/v1.1/system/user
```

### 10.1.5 确定 HTTP Method

在 REST 中，HTTP Method 常常对应以下含义：

- POST，代表增加资源；
- PUT，代表更改资源，客户端提供需完整的资源属性；
- GET，代表查询资源；
- PATCH，更新资源，客户端提供仅需要更改的资源属性；
- DELETE，通常用于删除资源；
- HEAD，类似 GET，但仅仅只有 HTTP 头信息，头信息包含了需要查找的信息；
- OPTIONS，用于获取 URI 所支持的方法，响应信息会在 HTTP 头中包含一个名为“Allow”的头，值是所支持的方法，如“GET、POST”。

> 在业务系统中，删除往往并不是指的物理删除，而是逻辑删除，资源通常仍然在数据库中，只是状态设置为删除状态。

比如新增用户：

```
POST http://192.168.0.1/api/v1.1/system/user
```

查询用户 id 为 451：

```
GET http://192.168.0.1/api/v1.1/system/user/451
```

查询所有用户：

```
GET http://192.168.0.1/api/v1.1/system/user
```

如果有翻页，可以在后面增加类似 offset、limit 参数，比如：

```
GET http://192.168.0.1/api/v1.1/system/user?offset=1&limit=20&sortBy=
name&sortOrder=desc
```

更新用户 id 为 451 的用户：

```
PUT http://192.168.0.1/api/v1.1/system/user/451
```

删除用户 id 为 451 的用户：

```
DELETE http://192.168.0.1/api/v1.1/system/user/451
```

> 可以为资源标识添加后缀，使得 REST 代码更加容易阅读，比如：
>
> ```
> GET http://192.168.0.1/api/v1.1/system/user/451.json
> ```
>
> 返回 id 为 451 的用户信息，返回格式是 JSON。
>
> 现在也有一种设计 REST URI 的方式，把操作也放到 URI 中，HTTP 方法主要采用 GET 和 POST，这样的好处同样是易于阅读。

## 10.1.6 确定 HTTP Status

服务器向客户端返回 HTPP Status 以表示操作是否成功，常用的如下：

- 200,OK，用户请求成功，如查询数据成功返回。
- 400，错误的的请求，在第 3 章中，URI 匹配上 Spring Boot 中的 Controller，但方法参数匹配错误，就会抛出错误。
- 404 NOT Found，用户发出的请求针对的资源不存在，通常是 Spring Boot 中的 Controller 没有匹配上 URI，或者匹配上了 Controller 方法，但渲染的视图不存在。
- 405，用来访问本页面的 HTTP Method 不被允许，比如通过 HTTP GET 方式访问了一个@PostMapping 的 Controller 方法。
- 406，表示无法使用请求的内容特性来响应请求的资源，比如在 Spring Boot 中，请求后缀以 html 结尾，但同时请求的 HTTP 头中又包含了 Accept:application/json。
- 500，服务器内部错误，无法完成请求，通常是 Controller 抛出的异常。

## 10.1.7 REST VS. WebService

WebService 是一种曾经流行的基于 HTTP 的接口方式，它的初衷现在看起来仍然是那么美好。比如：

- 建立在 SOAP 协议上，SOAP 是一种功能完备的消息交换协议。
- WSDL，WebService 描述语言，能描述 WebService 提供的服务名称、参数、调用协议等，通过 WSDL 还能生成客户端调用代码。
- WS-*，一系列与 WebService 相关的辅助规范。
- 异构系统之间一种互相调用方式。这在早期异构系统之间是一种优势，然而现在基于 Web 应用的平台，REST 也具备这个优点。

然而，美好的事情也并不全是看上去的那个样子，REST 和 WebService 主要的差别就是前者是一种轻量级的架构，而后者是一种重量级架构。前者既适合终端到服务的调用，系统内部子系统的互相调用，也适合不同公司之间的系统互相调用，而 WebService 较为适合不同公司之间系统的调用，这是因为：

- SOAP 协议过于复杂，是个重量级协议，SOAP 协议基于 XML，本来是要代替更早的 XML-RPC 协议，但自身却越来越复杂，开发一个 WebService 服务实现要比开发 REST 慢很多。

- WS-* 协议的复杂性，只有一些商业机构实现了 WS-*的规范。
- 在传输数据上，JSON 比 XML 更为流行，XML 数据封装，虽然数据表现能力强，但影响系统性能。比如 XML 解析曾经出现了一代比一代更好的方式，如 DOM、SAX、XML Sta，也验证了 XML 在性能方面确实不理想。
- XML 在网络之间传输的数据量也会大一些，不如 JSON 简单。

尽管 WebService 有上述缺点，但 WebService 仍然是一种流行的大型系统之间的交互方式，WebService 具备的一些功能，比如 WSDL，REST 并没有做出规定，本章最后会介绍 Swagger，这是一种 REST 辅助工具，用来增强 REST 应用。

> WS-* 是指一系列与 WS 相关的规范，比如：
>
> - WS-Security，安全相关，还有 WS-Trust。
> - WS-Addressing，访问调用相关规范。
> - WS-BPEL，业务处理规范。
> - WS-AtomicTransaction，事务相关，还有 WS-Transaction。
> - WS-Management，管理相关规范。

## 10.2 Spring Boot 集成 REST

### 10.2.1 集成 REST

只要 spring-boot-starter-web 依赖在 pom 中，即自动支持 REST：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

### 10.2.2 @RestController

注解@RestController 用于描述 REST 服务，相当于@Controller 和@ResponseBody 的组合。以下两个例子是等价的：

```
@RestController
```

```
@RequestMapping("/api/v1")
public class OrderApiCrontroller {

  @GetMapping("/order/{orderId}")
  public Order getOrder(@PathVariable String orderId) throws Exception{
    .....
  }
}

@Controller
@RequestMapping("/api/v1")
public class OrderApiCrontroller {

  @GetMapping("/order/{orderId}")
  public @ResponseBody Order getOrder(@PathVariable String orderId) throws
Exception{
      .....
  }
}
```

> REST 架构只是一种架构风格而不是特殊的一种技术，即使没有使用@RestController 注解，你也能完成 REST 架构。

为了测试 REST 服务，我们可以使用 curl 命令。如果是 Linux 和 Mac 系统，则自带 curl，如果是 Windows 系统，安装了 Git 后，也会自带 curl，否则得自行安装，可以参考第 3 章关于 curl 的介绍。

对于如下 REST 接口：

```
@PostMapping("/order")
public String addOrder(@RequestBody Order order) throws Exception{
  return "{success:true,message:\"添加成功\"}";
}
```

可以使用以下命令进行测试：

```
>curl -XPOST '127.0.0.1:8080/api/v1/order' -H 'Content-Type:
application/json' -d'
{
```

```
    "id" : "001",
    "name":"订单"
  }
  '
```

删除接口：

```
@DeleteMapping("/order/{orderId}")
public String cancelOrder(@PathVariable String orderId) throws Exception{
  return "{success:true,message:\"订单取消成功\"}";
}
```

> 绝大部分业务系统，删除并非代表真正的物理删除（从数据库物理删除），往往只是状态的变化。

可以使用以下命令进行测试：

```
>curl -XDELETE '127.0.0.1:8080/api/v1/order/100'
```

## 10.2.3 REST Client

前面讲到如何在 Spring Boot 中提供 RESTful 服务，下面介绍系统之间如何发起 REST 请求。

Spring Boot 提供了 RestTemplate 来辅助发起一个 REST 请求，默认通过 JDK 自带的 HttpURLConnection 来作为底层 HTTP 消息的发送方式，使用 JackSon 来序列化服务器返回的 JSON 数据。

### 10.2.3.1 RestTemplate

RestTemplate 是核心类，提供了所有访问 REST 服务的接口，尽管实际上可以使用 HTTP Client 类或者 java.net.URL 来完成，但 RestTemplate 提供了 RESTful 风格的 API。

RestTemplate 有 6 个主要的方法，分别对应于 RESTful 的 6 个主要的 HTTP Method，如下表所示。

| HTTP Method | Java API |
|---|---|
| DELETE | delete |
| GET | getForObject，getForEntity |
| HEAD | headForHeaders |

续表

| | |
|---|---|
| OPTIONS | optionsForAllow |
| POST | postForObject，postForLocation |
| PUT | put |
| 其他 | exchange（通用） |

Spring Boot 提供了 RestTemplateBuilder 来创建一个 RestTemplate。应用可以通过以下代码来创建一个 RestTemplate 实例：

```
@Autowired
RestTemplateBuilder restTemplateBuilder;
public void foo(){
  RestTemplate client = restTemplateBuilder.build();
}
```

使用 RestTemplate 来查询订单和创建订单：

```
/*一个测试类*/
@Controller
@RequestMapping("/test")
public class RestClientTestCrontroller {

  @Value(value = "${api.order}")
  String base ;
  @Autowired
  RestTemplateBuilder restTemplateBuilder;

  @GetMapping("/get/{orderId}")
  public @ResponseBody Order testGetOrder(@PathVariable String orderId)
throws Exception{
    RestTemplate client = restTemplateBuilder.build();
    String uri = base+"/order/{orderId}";
    // 核心代码
    Order order = client.getForObject(uri, Order.class,orderId);
    return order;
  }
}
```

base 是在配置文件 application.properties 中配置的订单 API 地址：

```
api.order=http://127.0.0.1:8080/api/v1
```

代码首先构造 RestTemplate，然后调用 getForObject，此方法接受三个参数，第一个参数是 URI 模板，第二个参数是期望返回的对象，后面是 URI 模板对应的参数列表。参数列表既可以是数组，也可以是 Map，以上代码又可以写成：

```
Map map = new HashMap();
map.put("orderId",orderId);
Order order = client.getForObject(uri, Order.class,map);
```

如果还想获取返回的 HTTP 头相关信息，可以调用 client.getForEntity，此方法返回 ResponseEntity，包含了头信息：

```
ResponseEntity<Order> responseEntity = client.getForEntity(uri,
Order.class, orderId);
Order order = responseEntity.getBody();
HttpHeaders headers = responseEntity.getHeaders();
```

添加订单可以使用 postForObject 方法，此方法接受三个参数，第一个是 URI，第二个是 Post 参数，可以是 HttpEntity，或者是某个 POJO 对象，POJO 对象在这种情况下会自动转成 HttpEntity，第三个是期望返回的类型，这个例子中期望返回的类型是 String。

```
@GetMapping("/addorder")
public @ResponseBody String testAddOrder() throws Exception{
  RestTemplate client = restTemplateBuilder.build();
  String uri = base+"/order";
  Order order = new Order();
  order.setName("test");
  String ret = client.postForObject(uri, order, String.class);
  //{success:true,message:"添加成功"}
  return ret;
}
```

或者使用 HttpEntity：

```
HttpEntity<Order> body = new HttpEntity<Order>(order);
```

```
String ret = client.postForObject(uri, body, String.class);
```

使用 HttpEnity 的好处是可以提供额外的 HTTP 头信息。

如果期望返回的类型是一个列表，如 List，不能简单调用 xxxForObject，因为存在泛型的类型擦除，RestTemplate 在反序列化的时候并不知道实际反序列化的类型，因此可以使用 ParameterizedTypeReference 来包含泛型类型，代码如下：

```
RestTemplate client = restTemplateBuilder.build();
// 根据条件查询一组订单
String uri = base+"/orders?offset={offset}";
Integer offset = 1;
// 无参数
HttpEntity body = null;
ParameterizedTypeReference<List<Order>> typeRef = new
ParameterizedTypeReference<List<Order>>() {};
ResponseEntity<List<Order>> rs = client.exchange(uri, HttpMethod.GET, body,
typeRef, offset);
List<Order> order = rs.getBody();
```

注意到 typeRef 定义是用{}结束的，这里创建了一个 ParameterizedTypeReference 子类，依据在类定义中的泛型信息保留的原则，typeRef 保留了期望返回的泛型 List。

exchange 是一个基础的 REST 调用接口，除了需要指明 HTTP Method，调用方法同其他方法类似。

> 除了使用 ParameterizedTypeReference 来保留泛型信息，也可以通过 getForObject 方法先映射成 String，然后通过 ObjectMapper 来转为指定类型，可以参考第 3 章 Jackson 来了解。

#### 10.2.3.2 定制 RestTemplate

创建一个配置类实现 RestTemplateCustomizer 接口的 customize 方法：

```
@Configuration
public class RestConf implements RestTemplateCustomizer {

    public void customize(RestTemplate restTemplate) {

        SimpleClientHttpRequestFactory jdkHttp =
(SimpleClientHttpRequestFactory)restTemplate.getRequestFactory();
```

```
        jdkHttp.setConnectTimeout(1000);
    }
}
```

customize方法会定制RestTemplate，上面的代码设置链接超时时间为1000毫秒。Spring Boot因为默认使用了JDK的URLConnection作为底层的HTTP工具，如果想使用了OkHttp，需要添加以下依赖：

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>3.8.1</version>
</dependency>
```

上面的代码应该是：

```
public void customize(RestTemplate restTemplate) {
  OkHttp3ClientHttpRequestFactory okHttp =
(OkHttp3ClientHttpRequestFactory)restTemplate.getRequestFactory();
  okHttp.setReadTimeout(5000);
  okHttp.setWriteTimeout(3000);
}
```

## 10.3 Swagger UI

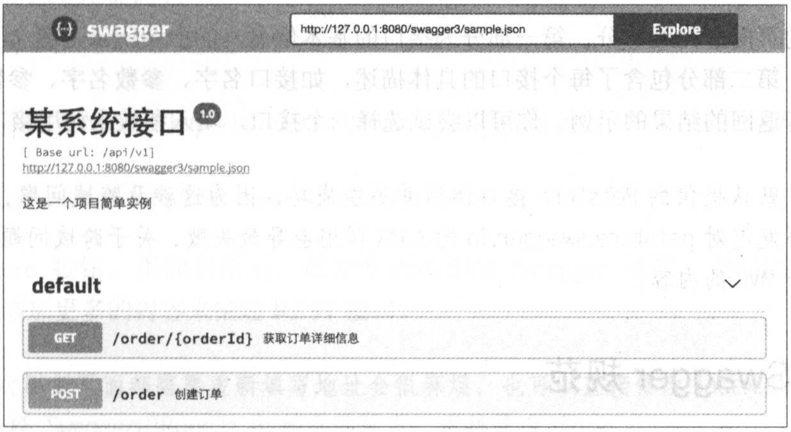
前面说过，WS提供了WSDL来描述提供的WS调用、参数，还有一系列辅助工具用来生成WS客户端代码和测试WebService。REST也有一个工具Swagger来完成类似功能，可以通过Swagger规范来描述RESTFul接口，通过Swagger UI来显示和测试RESTFul接口。

> 本书描述的是Swagger3.0的内容，与Swagger2.0的内容有较大差别。接口描述在3.0中通过Swagger规范（一个JSON文件)来描述，Swagger2.0是通过在接口中提供一系列注解来描述的。本书将介绍Swagger3.0的内容。

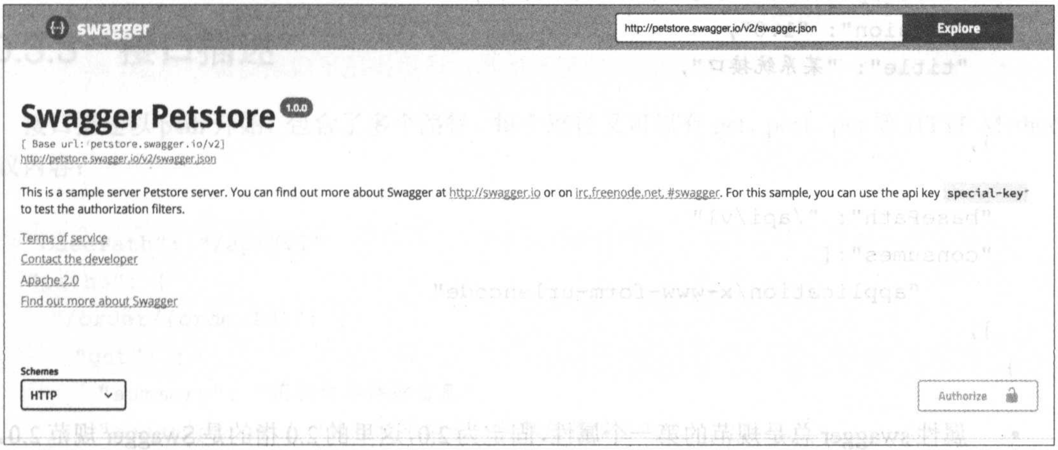
### 10.3.1 集成Swagger

Swagger提供了一组静态页面，可以在Spring Boot应用中集成这些静态页面，直接访问静

态页面，并打开指定的 Swagger 规范，就可以显示 RESTFul 接口，如下图所示。




- 进入 Swagger 官网，选择 Swagger UI 产品，进入并选择下载。
- 页面提示会进入 GitHub。
- 在 GitHub 中，进入该项目的 Release 页面，选择一个最新的版本下载，在写本书的时候，最新版本是 swagger-ui 3.0.8。
- 下载 3.0.8，解压后，找到 dist 目录，我们仅仅需要这一部分即可。在工程中创建 statics 目录，并且创建 swagger3 目录，复制刚才 dist 目录下的所有文件到 swagger3 下面。
- 访问 http://127.0.0.1:8080/swagger3/index.html，这时候能看到如下页面：




该页面加载的时候，会自动打开一个 swagger 接口规范文档，如上图右上角所示，

http://petstore.swagger.io/v2/swagger.json。你可以在浏览器中打开此链接，获取 Swagger Petstore 的接口规范作为后续参考。

打开后的页面分为两部分，第一部分为接口的基本信息，如上图所示，包含了项目名称、描述等信息；第二部分包含了每个接口的具体描述，如接口名字、参数名字、参数类型、是否必填等，还有返回的结果的示例。你可以尝试选择一个接口，填入参数来调用接口。

> **注意：**默认提供的 Petstore 接口调用并不能成功，因为这涉及跨域问题，在 localhost 环境下发起对 petstore.swagger.io 的 AJAX 调用会导致失败，关于跨域问题，请参考第 3 章关于 MVC 的内容。

## 10.3.2 Swagger 规范

Swagger 规范的内容较多，接口描述对使用者非常友好，本书为了简化介绍 Swagger，只重点介绍足以描述和测试 RESTFul 接口的部分内容。

Swagger 规范是一个 JSON 格式的文件，包含项目基本信息及具体接口描述信息，可以在 static/swagger3 下创建一个 sample.json 文件，我们将在本节逐渐完善。

项目描述、访问的接口的地址：

```
{
  "swagger": "2.0",
  "info": {
    "description": "这是一个项目简单实例",
    "version": "1.0",
    "title": "某系统接口",

  },

  "basePath": "/api/v1"
  "consumes":[
      "application/x-www-form-urlencode"
  ],
}
```

- 属性 swagger 总是规范的第一个属性，固定为 2.0，这里的 2.0 指的是 Swagger 规范 2.0。
- info 描述了一个项目基本信息。

- basePath 指的是 RESTFul 接口的实际地址，以上是/api/v1，则 REST 接口的地址则是 127.0.0.1:8080/api/v1。
- consumes 指提交的内容是表单。

这个时候，可以再次访问 http://127.0.0.1:8080/swagger3/index.html，并且在页面填写规范地址：

```
http://127.0.0.1:8080/swagger3/sample.json
```

点击 explore 按钮，页面刷新后，就能看到添加的 Swagger 规范。我们随后将陆续向 sample.json 中添加更多的内容来描述 REST 接口。

> 如果每次刷新页面都需要重新填写地址会很麻烦，也可以直接修改 index.html 文件，找到 JS 片段 SwaggerUIBundle 初始化的地方，直接修改 URL：
>
> ```
> <script>
> window.onload = function() {
>  // Build a system
>  const ui = SwaggerUIBundle({
>  url: "http://127.0.0.1:8080/swagger3/sample.json",
>  dom_id: '#swagger-ui',
>  .....
>  })
>  }
> ```

## 10.3.3 接口描述

接口描述以 path 开始，包含了多个路径，每个路径又可以有 get、post、put 等 HTTP Method 协议内容：

```
"basePath": "/api/v1"
  "paths": {
    "/order/{orderId}": {
      "get": {
        "summary": "获取订单详细信息",
        "description": "传入订单编号，获取订单信息",
        "parameters": [
```

```
      {
        "name": "orderId",
        "in": "path",
        "description": "订单 id",
        "required": true,
      }
    ],
    "responses": {
      "200": {
        "description": "获取用户信息成功"
      }
    }
  }
 }
}
```

每个接口包含了以下信息：

- summary，接口主要功能的简要描述；
- description，接口详细描述；
- parameters，接口的参数，REST 参数在 Swagger 中分为四种类型，以上实例的参数类型是 path，也就是参数是从 path 中获取的，其他的还有 body、parameter 等，我们将在下一节详细描述。
- response，对应了 HTTP status 的提示信息，这里描述了成功的提示信息。

## 10.3.4 查询参数描述

查询参数，in 的值是 query，类似/order?offset=0，这里的 offset 就是 url 参数，在 Swagger 中，表示如下：

```
"parameters": [
  {
    "name": "offset",
    "in": "query",
    "description": "查询起始位置",
```

```
      "required": true,
    }
  ]
```

### 10.3.5 URI 中的参数

URI 模板中的参数，如/order/{orderId}，in 的值使用 path：

```
"parameters": [
  {
    "name": "orderId",
    "in": "path",
    "description": "订单 id",
    "required": true,
  }
],
```

### 10.3.6 HTTP 头参数

设置 HTTP 头中的参数，in 的值使用 header：

```
"parameters": [
  {
    "name": "X-Request-ID",
    "description": "",
    "in": "header"
  }
]
```

### 10.3.7 表单参数

使用 application/x-www-form-urlencoded 提交的参数，in 的值使用 formData：

```
"parameters": [
  {
    "name": "orderName",
    "description": "",
    "in": "formData"
```

```
    }
]
```

## 10.3.8 文件上传参数

需要增加 type 属性，值是 file：

```
"parameters": [
  {
    "name": "orderName",
    "description": "",
    "in": "formData",
    "type":"file"
  }
]
```

## 10.3.9 整个请求体作为参数

通常是将 JSON 数据作为 HTTP 内容体发送到服务器端：

```
{
  "/order": {
    "post": {
     "summary": "创建订单",
      "description": "创建一个新的订单",
        "parameters": [
          {
            "name": "order",
            "in": "body",
            "description": "包含订单信息的 JSON",
            "required": true,
            "schema": {
              "$ref": "#/definitions/order"
            }

          }
        ],
        "responses": {
          "200": {
```

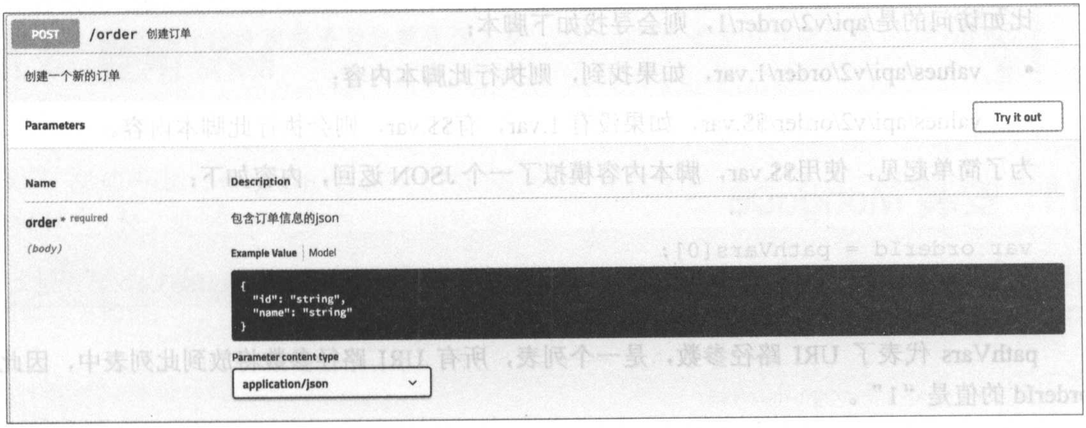
```
                    "description": "创建订单成功"
                }
            }
        }
    }
}
```

既然参数类型是 body，name 是可选的属性，schema 是可选项，说明了 body 的格式，$ref 表示 body 格式规范定义在其他地方 “/definitions/order”，参考了 definitions 下的 order 属性，内容如下：

```
"definitions": {
  "order": {
    "type": "object",
    "properties": {
      "id": {
        "type": "string",

      },
      "name": {
        "type": "string",
      }
    }
  }
}
```

如果打开 Swagger UI，会有以下接口展示：



点击“Try it Out”按钮，Swagger UI 会自动生成一个符合定义的 Schema 的 JSON 样例数据，你可以修改后执行测试的 REST 服务。

## 10.4 模拟 REST 服务

Swagger UI 描述了 REST 服务，以及提供客户端调用工具，如果你的 Spring Boot 应用刚好选择了 Beetl，则可以在一定程度上模拟 REST 服务，而不必要等待 REST 服务开发完毕。

比如，要为订单接口创建新的版本，如/api/v2，不必先完成代码，可以通过 WebSimulate 来模拟实现，创建以下 Controller：

```
@RestController
@RequestMapping("/api/v2/")
public class OrderApi2Crontroller {

  @Autowired
  WebSimulate webSimulate;
  @RequestMapping("/**")
  public void simluateJson(HttpServletRequest request,HttpServletResponse
response){
    webSimulate.execute(request, response);
  }
}
```

以上代码所有以/api/v2 开头的访问，都会交给 simulateJson 方法处理，WebSimulate 将代理这一请求处理，会在 resources/templates/values 目录下寻找匹配的 Beetl 脚本文件执行并返回。

比如访问的是/api/v2/order/1，则会寻找如下脚本：

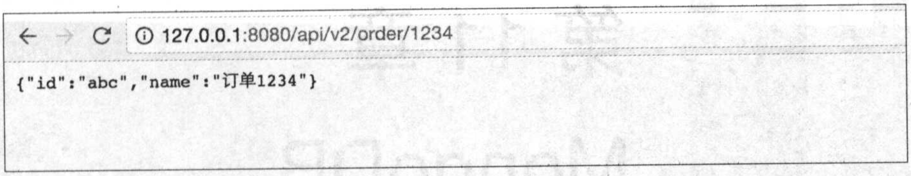
- values/api/v2/order/1.var，如果找到，则执行此脚本内容；
- values/api/v2/order/$$.var，如果没有 1.var，有$$.var，则会执行此脚本内容。

为了简单起见，使用$$.var，脚本内容模拟了一个 JSON 返回，内容如下：

```
var orderId = pathVars[0];
var json = {"id":"abc",name:"订单"+orderId};
```

pathVars 代表了 URI 路径参数，是一个列表，所有 URI 路径参数将放到此列表中，因此 orderId 的值是“1”。

json 是一个特殊的变量，WebSimulate 将返回此变量值给 REST 客户端。访问效果如下图所示。




> 使用 WebSimulate 能尽早地给 REST 客户端提供接口实现。关于 WebSimulate，请参考第 4 章视图技术，以及 Beetl 的官网 ibeetl.com。
>
> 可以在 OrderApi2Controller 上逐步添加业务实现来替代模拟实现 Spring MVC 的 URL 路径映射，路径带通配符的匹配优先级低于具体的 URL 路径匹配。

# 第 11 章 MongoDB

chapter 11

MongoDB 由 C++语言编写，是一个基于分布式文件存储的开源数据库系统，支持的数据结构为 BSON 格式，类似 JSON 的一种格式，因此可以存储非常复杂的数据，具有以下特点：

- 支持各种编程语言，Java、C++、PHP、C#、Python 等。
- 面向文档存储，文档格式是类似 JSON 的 BSON 格式。
- 提供丰富的查询功能，支持对数据建立索引。
- 模式自由，不需要事先定义文档格式，可以任意改变文档格式。

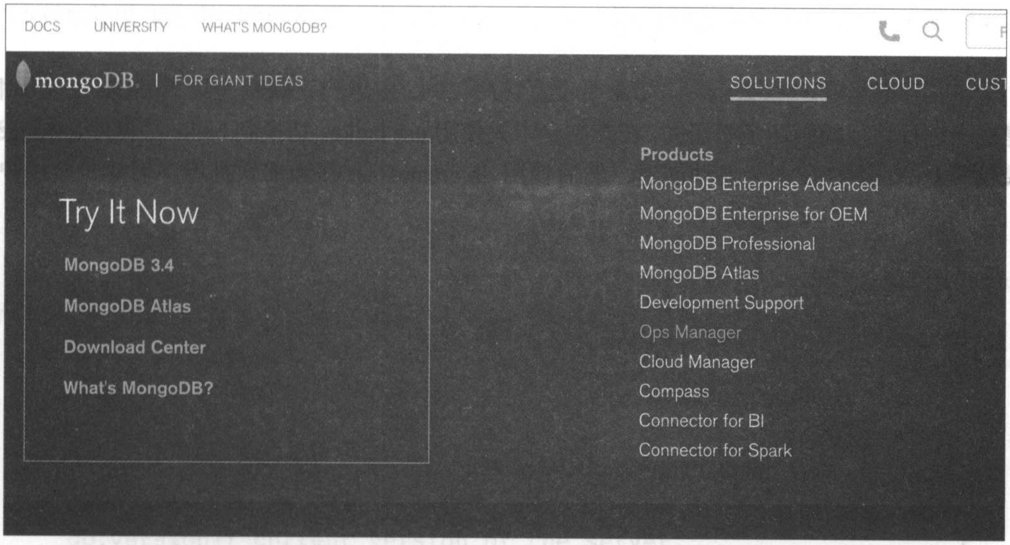
> 在处理 JSON 文档上，PostgreSQL 数据库恰好也有类似的功能支持 JSON Type，如果你的 Spring Boot 应用刚好在使用 PostgreSQL，那么还是建议优先使用 PostgreSQL 的 JSON 功能。

## 11.1 安装 MongoDB

安装 MongoDB 比较简单，本书写作时，MongoDB 的版本刚好是 MongoDB3.4。首先进入官网页面：

```
https://www.mongodb.com/
```

点击 solutions，在出现的列表中下载 MongoDB3.4，然后解压即可：




解压后，进入 bin 目录，有以下常用命令：

- mongod，启动 MongoDB 数据库。通常需要指定一个数据存放的目录，默认是/data/db。如果启动的时候没有创建/data/db，则会启动失败。可以通过—dbpath 参数指定特定的目录，参数详情可以键入./mongod —help。
- mongo，启动一个 Shell ./mongo，通过 shell 可以对 MongoDB 进行增删改查等操作。
- mongodump，备份数据库。
- mongorestore，恢复数据库。

## 11.2 使用 shell

在编写 Spring Boot 代码操作 MongoDB 前，使用 Shell 来操作 MongoDB，对数据进行增删改查。如前所述，MongoDB 是面向文档的数据库，文档使用 JSON 格式，准确地说，应该是 BSON，一种二进制压缩、带类型的 JSON 格式。

进入 bin 目录，运行 mongod 启动 MongoDB 数据库，运行 ./mongo，便进入 shell，会看到类似以下输出：

```
MongoDB shell version v3.4.4
connecting to: mongodb://127.0.0.1:27017
```

```
MongoDB server version: 3.4.4
......
>
```

输出信息包含了一些警告，比如需要设置访问控制。考虑到互联网大量 NoSQL 安全漏洞，创建账户进行访问控制还是必需的，在控制台中先使用 use baike，切换到 baike 数据库，MongoDB 如果检测 baike 不存在，会自动创建，然后使用 db.createUser 在当前数据库中创建一个用户：

```
>use baike
> db.createUser(
  {
    user: "test",
    pwd: "123%abc!",
    roles: [
      { role: "readWrite", db: "baike" }
    ]
  }
)
```

字段 user 和 pwd 分别表示用户的名称和密码，roles 代表了用户角色，readWrite 表示数据库读写权限，db 表示限定的数据库是 baike（百科）。

创建用户后，需要重新启动 mongod，并且使用—auth 参数，这样就打开了数据库验证功能。

再次使用 mongo 命令连接到 Mongo 服务器上，使用 db.auth 来登录：

```
>use baike
>db.auth("test","123%abc!")
1
```

控制台输出 1 表示创建成功。

> **注意：**本章后面的内容将简要介绍 MongoDB 的基本增删改查功能，不会介绍 MongoDB 的管理和运维方面的知识，读者可以查询官网文档说明来了解这些内容。

### 11.2.1 指定数据库

通过 use 命令可以指定使用的数据库，如果数据库不存在，则创建一个，以下命令从 baike 数据库切换到 test2 数据库：

```
>use test2
switched to db test2
```

可以通过“db”命令查看当前正在使用哪个数据库，通过键入 db.help()方法来获取更多关于 db 命令的内容：

```
>db.help()
DB methods:
    db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs
command [ just calls db.runCommand(...) ]
    db.auth(username, password)
    db.cloneDatabase(fromhost)
    .....
    db.shutdownServer()
    db.stats()
    db.version() current version of the server
```

## 11.2.2 插入文档

MongoDB 通过集合（Collection）来管理数据库，类似于数据库表，集合包含了多个文档（Document），文档类似于数据库表的记录。MongoDB 不需要显式地创建集合，可以直接给集合添加文档，比如给 baike 集合添加一个条目。

```
>db.baike.insert({_id:"springboot",desc:"快速分布式开发框架",
tag:["IT","Spring"],comment:{good:1256,bad:12}})
```

baike 是集合的名字，当不存在集合的时候会自动创建此集合。insert 方法用于以 JSON 格式插入一个文档。在本例中，设定 baike 文档包含了四个字段：

- _id，MongoDB 以下画线开头的字段都有特殊意义，表示文档主键，如果文档没有提供此主键，则系统自动生成一个 ObjectID 类型的主键。推荐在 Spring Boot 应用中，由应用来定义一个主键而不是由 MongoDB 生成。
- desc，baike 条目的描述。文档可以包含任意属性及任意文档。
- tag，baike 的 tag，用数组来保存，MongoDB 的字段类型可以是字符串、数字、boolean，或者日期等类型，也可以是文档类型。
- comment：此字段是一个 JSON 文档，分别包含点赞和踩的数量。

> ObjectID 是 MongoDB 的一种特殊类型，用 12 字节存储，分别是 4 字节时间戳，3 字节机器唯一标识，2 字节的进程标识，以及最后 3 字节的自增，ObjectID 转成可读的 16 进制看起来是这个样子：4e7020cb7cac81af7136236b。

### 11.2.3 查询文档

MongoDB 支持按照主键查询，也可以根据条件进行查询，支持类似 and or in 等操作。

使用 db.collection.find()查询所有文档，如查询所有百科词条：

```
>db.baike.find()
{ "_id" : "beetl", "desc" : "新一代模板语言", "tag" : [ "IT", "模板语言" ],
"comment" : { "good" : 56, "bad" : 1 } }
{ "_id" : "springboot", "desc" : "快速分布式开发框架", "tag" : [ "IT", "Spring" ],
"comment" : { "good" : 1256, "bad" : 12 } }
```

通过条件查询：

```
>db.baike.find({_id:"springboot"})
{ "_id": "springboot", "desc" : "快速分布式开发框架", "tag" : [ "IT", "Spring" ],
"comment" : { "good" : 1256, "bad" : 12 } }
```

查询点赞数量大于 1000 的条目：

```
> db.baike.find( { "comment.good" : { $gt: 1000 } } )
{ "_id" : "springboot", "desc" : "快速分布式开发框架", "tag" : [ "IT", "Spring" ],
"comment" : { "good" : 1256, "bad" : 12 } }
```

可以使用类似 SQL 的 and or in 来查询，以下查询使用 and 来查询，查询被赞超过 1000 或者小于 100 的条目：

```
> db.baike.find( { "comment.good": { $gt: 1000 },"comment.good":{$lt:100} } )
```

使用 or 查询：

```
>db.baike.find({$or: [ {"_id" : "springboot"},{_id:"beetl"}]})
```

使用 in 查询：

```
db.baike.find( { _id: { $in: [ "springboot", "beetl" ] } } )
```

常用的查询比较如下：

- $gt 相当于“>”，如 "comment.good": { $gt: 1000 }；
- $gte 相当于“>=”，如 "comment.good": { $gte: 1000 }；
- $lt 相当于“<”，如 "comment.good": { $lt: 1000 }；
- $lte 相当于<=，如 "comment.good": { $lte: 1000 }；
- $eq 相当于“==”，如 "comment.good": { $eq: 1256 }或者{"comment.good": 1256 }；
- $ne 相当于“!= ”，如 "comment.good": { $ne: 1256 }；
- $in 等同于 in，如 "tag": {$in:["IT","模板语言"]} ，查找所有包含“IT”和“模板语言”的 tag；
- $nin 等同于 not in。

模糊搜索：

```
>>db.baike.find({"desc":/.*框架.*/})
```

> MongoDB 也支持全文搜索，但中文支持得不是很好，而且搜索效率较低，全文搜索可以参考本书关于 Elasticsearch 的介绍。

## 11.2.4 更新操作

MongoDB 的更新操作类似关系型数据库，提供以下 API：

```
db.collection.updateOne(<filter>, <update>, <options>)
db.collection.updateMany(<filter>, <update>, <options>)
db.collection.replaceOne(<filter>, <replacement>, <options>)
```

filter 表示查询条件，比如对 Spring Boot 条目修改说明，使用 updateOne 方法：

```
>db.baike.updateOne({_id:"springboot"},{$set: { "desc" : "基于 spring 的分
布式开发框架" } });
```

replaceOne 与 updateOne 的区别在于前者是替换整个文档，而后者是更新部分文档。这两

个方法都只替换匹配的第一条记录，如果有多条记录匹配，可以使用 updateMany。

options 有如下属性可以控制更新特性：

- upsert，默认为 false，如果设置成 true，未匹配的文档会 insert 到数据库中。
- writeConcern，对写入进行配置，包含以下属性。
  - w，默认是 {w:0}，表示写入后不需要数据库发送确认（ACK），这样性能很高；{w:1}，数据写入到主库就向客户端发送确认；{w:"majority"}，数据写入到所有节点后向客户端发送确认。
  - wtimeout，当 w 大于 1 的时候，设置一个等待时间，单位为毫秒，如果超过这个时间，即使数据成功写入到库中，客户端也得到一个错误信息。

### 11.2.5 删除操作

按照条件删除文档，以下为没有指定条件，删除所有文档：

```
db.baike.deleteMany({})
```

按照条件删除一个文档：

```
db.baike.deleteOne({_id:"springboot"})
```

## 11.3 Spring Boot 集成 MongoDB

Spring Boot 引入 spring-boot-starter-data-mongodb 来集成 MongoDB，以下是在 pom 中添加的内容：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

还需要在配置文件 application.properties 中配置 MongoDB 的链接地址：

```
spring.data.mongodb.uri=mongodb://127.0.0.1:27017/baike
```

27017 是 MongoDB 默认的端口，test 是内置的数据库。如果 MongoDB 使用了--auth 选项启

动，则需要使用用户登录，使用以下方式进行链接：

```
spring.data.mongodb.uri=mongodb://test:123%25abc!@127.0.0.1:27017/baike
```

> **注意：**“%”字符在 URI 中使用%25，这是因为 URI Encode 的结果。如果你不清楚你的密码对应的 URI 的编码是什么，可以从网上找“在线 URI 编码”，会有很多在线工具帮助你转换。

重启 Spring Boot 应用，能看到以下提示信息，表示成功：

```
org.mongodb.driver.connection                : Opened connection
[connectionId{localValue:1, serverValue:14}] to 127.0.0.1:27017
org.mongodb.driver.cluster                   : Monitor thread successfully
connected to server with description ServerDescription{
```

# 11.4 增删改查

本章假定文档对应的对象如下：

```
public class Baike {
  // 百科关键字，id 将对应 MongoDB 的_id
  private String id;
  private String desc;
  private List<String> tag = new ArrayList<String>();
  private Comment comment = null;
  private Date crateDate =null;
  private Date updateDate = null;
}
```

Spring Boot 使用 MongoTemplate 作为核心来进行增删改查操作，提供了 insert、findById、find、findAndModify、findAndRemove 等方法。MongoTemplate 可以自动注入到 Spring 管理的 Bean 中，如 Controller 或者 Service 中。

## 11.4.1 增加 API

MongoTemplate 提供了 insert 和 save 两个方法来新增文档或文档对象，前者新增一个文

档，后者新增或者更新文档对象。对象名字的首字母小写变成 MongoDB 集合的名称，如 Baike 对象在 MongoDB 中集合名字是 baike。

```
@GetMapping("/addbaike")
public Baike addDict(Baike baike){
  baike.setCrateDate(new Date());
  mongoTemplate.insert(baike);
  return baike ;
}
```

以上代码向 MongoDB 中新增了一个百科文档，考虑到测试的方便，使用了 GetMapping，如果要使用 PostMapping，可以通过 curl 命令来进行测试。在浏览器中输入：

```
http://127.0.0.1:8080/addbaike?id=beetlsql&desc=dao 工具&tag[0]=
IT&comment.bad=1
```

以上请求通过 Spring MVC 自动构造了 baike 对象实例，并通过 mongoTemplate 插入到数据库。

insert 方法也允许指定一个集合名字，如：

```
mongoTemplate.insert(baike,"baike");
```

> 为了测试方便，使用了@GetMapping 来添加或者修改文档，实际上使用@PostMapping 更好。

## 11.4.2 根据主键查询 API

mongoTemplate 提供了 findById，用于根据主键查找文档，并映射到 Java 对象：

```
@GetMapping("/baike/{name}")
public Baike findUser(@PathVariable String name){
  Baike dict = mongoTemplate.findById(name, Baike.class);
  return dict ;
}
```

通过浏览器输入：

```
http://127.0.0.1:8080/baike/beetlsql
```

会有以下输出：

```
{"id":"beetlsql","desc":"dao 工具","tag":["IT"],"comment":{"good":0,"bad":1},
"crateDate":1498831559404,"updateDate":null}
```

### 11.4.3 查询 API

通过 Criteria 类构造查询条件，调用 find 方法查找。比如查询 comment 的 bad 属性大于一定数量的 Baike 对象：

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query.query;

@GetMapping("/querybad/{bad}")
public List<Baike> queryBad(@PathVariable int bad){
  Criteria criteria = where("comment.bad").gt(bad);
  List<Baike> list = mongoTemplate.find(query(criteria), Baike.class);
  return list ;
}
```

Criteria 对象包含了很多方法来构造查询条件，比如构造一个 and 条件：

```
Criteria criteria = where("comment.bad").gt(bad);
Criteria criteria2 = where("comment.good").lt(good);
List<Baike> list = mongoTemplate.find(query(criteria.andOperator(criteria2)),
Baike.class);
```

查询点赞个数小于 good，负面评价大于 bad 的所有 Baike。

query 方法构造了一个 Query 对象，还可以调用 limit 方法限定返回的个数，调用 skip 方法从指定的行数开始。以下是一个翻页查询：

```
@GetMapping("/baike/tag/{tag}/{pageNum}")
public List<Baike> findBaike(@PathVariable String tag,@PathVariable int
pageNum){
  Criteria criteria = where("tag").in(tag);
  Query query = query(criteria);
```

```
    // 查询总数
    long totalCount = mongoTemplate.count(query, Baike.class);
    // 每页个数
    int numOfPage = 10;
    // 计算总数
    long totalPage = totalCount%numOfPage==0?(totalCount/numOfPage):
(totalCount/numOfPage+1);

    int skip= (pageNum-1)*numOfPage;
    query.skip(skip).limit(numOfPage);
    List<Baike> list = mongoTemplate.find(query, Baike.class);
    // 构造一个 Page 对象，包含总数、当前页数，以及查询结果集，这里忽略
    return list;
}
```

## 11.4.4 修改 API

除了 save 方法能修改文档，mongoTemplate 通过 update 方法也可直接修改文档，API 如下：

```
  public UpdateResult updateFirst(Query query, Update update, Class<?>
entityClass){}
  public UpdateResult updateMulti(Query query, Update update, Class<?>
entityClass){}
```

Query 类可以参考 11.4.3 节，Update 对象用于构造 MongoDB 的 update 语句。updateFirst 更新符合条件的第一条记录，updateMulti 更新所有记录。

```
@GetMapping("/baike/tag/{tag}")
public @ResponseBody String addOne(@PathVariable String tag){
  Criteria criteria = where("tag").in(tag);
  Update update = new Update();
  // 自增
  update.inc("comment.good", 1);
  UpdateResult result = mongoTemplate.updateMulti(query(criteria), update,
Baike.class);
  return "成功修改 "+result.getModifiedCount() ;
}
```

通过浏览器输入以下 URL，所有 tag 包含 IT 的点赞数自增一个：

```
http://127.0.0.1:8080/baike/tag/IT
```

## 11.4.5 删除 API

mongoTemplate 提供 remove 方法来删除文档。

```
public DeleteResult remove(Object object){}
public DeleteResult remove(Query query, Class<?> entityClass){}
```

比如删除 Baike 对象：

```
@GetMapping("/deletebaike")
public Baike deleteDict(String id){
  Baike baike = new Baike();
  baike.setId(id);
  mongoTemplate.remove(baike);
  return baike ;
}
```

> 通常，无论是在 MongoDB 中的文档还是数据库的数据，都要尽量避免物理删除。建议为 Baike 增加一个 status，用于表示是否删除，通过 11.4.4 节的 update 方法来进行逻辑删除。

## 11.4.6 使用 MongoDatabase

Spring Boot 的 mongoTemplate 底层使用了 mongodb-driver-3.4.2.jar 驱动，如果你更喜欢使用底层驱动来访问 MongoDB，可使用 execute 方法来直接操作 MongoDB。调用 MongoTemplate 的方法如下：

```
public <T> T execute(DbCallback<T> action)
```

DbCallback 提供了底层驱动的 MongoDatabase 类来访问 MongoDB，以下是完成 11.4.2 节根据主键查询 API 的另外一种实现方式：

```
@GetMapping("/baike/{name}")
public Baike findUser(@PathVariable String name) {
  final String id = name;
  Baike baike = mongoTemplate.execute(new DbCallback<Baike>() {
    public Baike doInDB(MongoDatabase db) throws MongoException,
DataAccessException {
      // 获得集合
      MongoCollection<Document> collection = db.getCollection("baike");
      // 查找文档
      Document doc = collection.find(new Document("_id", id)).first();
      // 将 Document 对象转化成 Baike 对象
      Baike baike = new Baike();
      baike.setDesc(doc.getString("desc"));
      Comment comment = new Comment();
      Document docComment = doc.get("comment", Document.class);
      comment.setBad(docComment.getInteger("bad"));
      comment.setGood(docComment.getInteger("good"));
      baike.setComment(comment);
      return baike;
    }
  });
  return baike;
}
```

这段代码较长，因为篇幅限制，只完成 Document 到 Baike 对象的部分转化，关于底层驱动的详细说明超出了本书的范围。对于上述代码，简要说明一下：

- MongoDatabase，代表 MongoDB 数据库。
- MongoCollection，代表数据库的某一集合，提供了大量 CRUD 方法来操作集合。
- Document，代表集合中的文档，在 Java 中类似 HashMap 结构。
- find 方法返回了 MongoCursor 对象，也可以遍历这个对象，如下：

```
MongoCollection<Document> collection = db.getCollection("baike");
MongoCursor<Document> cursor =collection.find(new Document("_id", id)).
iterator();
try {
  while (cursor.hasNext()) {
```

```
    // 调用 Document 的 toJson 方法，返回 JSON 字符串
    System.out.println(cursor.next().toJson());
  }
} finally {
  cursor.close();
}
```

> 关于 Mongo Driver 的用法，可以参考 MongoDB 官网文档，地址是 https://docs.mongodb.com/ecosystem/drivers/.

## 11.4.7　打印日志

mongoTemplate 提供了大量 API 来访问 MongoDB，通过配置日志等级来了解访问细节，设置如下：

```
logging.level.org.springframework.data.mongodb=debug
```

可以看到控制台有类似以下输出：

```
o.s.data.mongodb.core.MongoTemplate          : find using query: { "tag" :
{ "$in" : [ "IT"]}} fields: null for class...

5] o.s.data.mongodb.core.MongoTemplate       : Calling update using query:
{ "tag" : { "$in" : [ "IT"]}} and update: { "$inc" : { "comment.good" : 1}} in
collection: baike
```

第一行日志了对应了 11.4.3 节查询 API 中查询 tag 包含 IT 的所有 Baike，第二行日志对应了 11.4.4 节查询 tag 包含 IT 并对 comment.good 自增 1 的更新操作。

# 第 12 章 Redis

Redis（REmote DIctionary Server）是一个开源（BSD 许可）、内存存储的数据结构服务器，可用作数据库来存储 Key-Value 数据，它支持字符串、哈希表、列表、集合、有序集合、位图、地理空间信息等数据类型，同时也可以作为高速缓存和消息队列代理。

Redis 与其他 NoSQL 相比，独特性在于支持复杂的数据结构，这些数据结构通常都与程序的数据机构一致，因此容易理解和使用。

Redis 在内存中存储数据，因此原则上，存放在 Redis 中的数据不应该大于内存容量，否则会因为操作系统虚拟内存导致性能降低。

## 12.1 安装 Redis

安装 Redis 比较简单，本书写作时 Redis 的版本是 Redis3.2.9。首先进入官网页面：

```
https://redis.io/
```

点击 Download，选择下载 3.2 版本，然后解压即可。

解压后，进入 src 目录，运行 make 进行编译：

```
>make
```

编译 Redis 大约需要一分钟，如果一切正常，会看到以下提示：

```
cc -O2 -Wall -DLUA_ANSI -DENABLE_CJSON_GLOBAL -DREDIS_STATIC=''  -c -o
ltablib.o ltablib.c
......
Hint: It's a good idea to run 'make test' ;)
```

可以再次运行 make test 以验证是否安装成功。

> Redis 主要运行在 Linux、BSD、Mac 中，也可以运行在 Solaris 中但不保证功能正确。
>
> 如果在 make 阶段报错，最有可能的是你的环境中没有安装 C 语言的编译环境，需要你安装 gcc，如何安装取决于你的具体操作系统。
>
> ```
> (Linux) yum install -y gcc
> (Mac)brew install gcc
> ```

Redis 安装完毕，在 src 目录下有以下两个常用命令：

- redis-server，启动 Redis 服务器，默认会监听 6379 端口；
- redis-cli，Redis 自带的客户端管理工具。

## 12.2 使用 redis-cli

Redis 一共有 14 个命令组、两百多个命令，本书不是专门介绍 Redis 的书，因此会结合常用的 Redis，以及后面涉及的 Spring Session、Spring Cache 来简单举例子。

进入 src 目录，运行./redis-server，进入 shell，会看到类似以下输出：

```
4027:M 13 May 19:29:34.613 # Server started, Redis version 3.2.9
4027:M 13 May 19:29:34.613 * The server is now ready to accept connections
on port 6379
```

重新打开一个终端，进入 src 目录，运行./redis-cl，进入 Redis 客户端管理工具：

```
localhost:src xiandafu$ ./redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

输入 ping 命令，用来检验 Redis 服务器是否正常运行，服务器返回 PONG 来应答服务器正常运行。

### 12.2.1 安全设置

由于互联网上出现了大量 NoSQL 数据库被不安全使用而导致的安全问题，因此使用 NoSQL 的第一步就是设置访问密码，打开 redis.conf 文件，添加一行密码：

```
requirepass Redis!123
```

以上配置设置了连接 Redis 的密码。进入 src 目录，再次启动服务器，并使用 redis.conf 文件：

```
./redis-server ../redis.conf
```

重新使用 redis-cli 连接，使用 ping 命令，会发现需要授权：

```
>127.0.0.1:6379> ping
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth "Redis!123"
```

使用 auth 命令来验证合法性，随后的操作才能成功。

### 12.2.2 基本操作

set 命令可以添加/覆盖一个字符串或者数字类型：

```
127.0.0.1:6379> set platform:info "simple infomation"
OK
127.0.0.1:6379> get platform:info
"simple infomation"
```

以上操作通过 set key value 来设置一个字符类型，然后通过 get 来获取其值，也可以通过 mget 来获取多个 key 的值，如：

```
127.0.0.1:6379> mget platform:version platform:info
```

```
1) "1"
2) "simple infomation"
127.0.0.1:6379>
```

Reids 或者 Spring Boot 中的 Key，通常都是包含逻辑上的命名空间，用符号“:”分开，比如 spring:session:xxxxxxx。

对于数字类型的字符串，还有以下命令可以对其进行算数操作：

- DECR/INCR，数字类型数据自减和自增。
- DECRBY/INCRBY，数字类型数据减去某个指定的整数或者增加某个指定整数。
- INCRBYFLOAT，数字增加一个浮点数，负数表示减去。

```
127.0.0.1:6379> set platform:version 1
OK
127.0.0.1:6379> incrby platform:version 2
(integer) 3
```

> 以上操作都是原子操作，不必担心多个客户端同时修改导致的并发问题，比如某个 Key 的值为 1，两个客户端同时自增这个 Key，结果一定是 3，而不是 2。
>
> mset 可以同时设置多个值，mget 可以同时获取多个值，这有助于减小网络操作延时。

## 12.2.3 keys

Redis 是 Key-Value 数据库，在 Redis 中，Key 是二进制数，因此字符串和图片都可以作为 Key，可以通过 keys 命令来查询 Redis 中所有的 Key：

```
127.0.0.1:6379> keys platform:*
1) "platform:info"
```

Key 后面可以用“*”或者“？”：

- Platform:*，匹配“platform:”开头的 Key；
- pl?tform，匹配 platform 或者 pletform；
- *，查询所有的 keys。

通过 exist 来判断 Key 是否存在：

```
>127.0.0.1:6379> exists platform:info
(integer) 1
```

返回 1 表示存在，0 表示不存在。

通过 del 命令删除 Key-Value，比如删除 platform:info：

```
127.0.0.1:6379> del platform:info
(integer) 1
```

返回 1 表示删除成功，0 表示失败。

> 事实上，exists和del命令后面可以有多个Key，用空格分开。操作结果是累计起来的结果。

可以设定 Key 的超时时间，当这个时间达到后，会被自动删除。以下是通过 expire 指定多少秒后 Key-Value 自动删除：

```
127.0.0.1:6379> expire platform 10
```

ttl 命令则用于查看 Key 的存活时间：

```
>127.0.0.1:6379> ttl platform
(integer) 7
```

表示 platform 还有 7 秒的存活时间。过了 7 秒后，再次用 get 命令操作，则返回 nil：

```
127.0.0.1:6379> get platform
(nil)
```

## 12.2.4 Redis List

Redis List 类型类似 Java 的 LinkedList，通过链表来完成，向其添加元素速度非常快，但按照索引方式获取元素比较慢。因此 List 结构适合那种大数据量，要求插入速度极快的场景。

rpush 可以将多个值放入 list 尾部，也可以理解为将值从 List 右边放入。lpush 则可以将多个值放到 list 头部，也可以理解为从 List 左边放入。使用 lrange 能从左到右显示指定范围的列表：

```
127.0.0.1:6379> rpush platform:history "2012-1-1" "2012-3-5"
(integer) 2
127.0.0.1:6379> lpush platform:history "2011-10-22"
(integer) 3
127.0.0.1:6379> lrange  platform:history  0 2
1) "2011-10-22"
2) "2012-1-1"
3) "2012-3-5"
```

lrange 的第一个参数是 key，后面两个参数是列表的范围，从列表末尾开始，因此上述 lrange 操作取出了 platform:history 的最后三个元素。

rpop 命令可以从列表尾部取出一个元素，lpop 则会取出列表的头一个元素：

```
127.0.0.1:6379> rpop platform:history
"2012-3-5"
127.0.0.1:6379> lpop platform:history
"2011-10-22"
127.0.0.1:6379> llen platform:history
(integer) 1
```

llen 用来返回 List 的长度，如上所示，列表原来的长度为 3，通过 rpop 和 lpop 分别取出 List 尾部和 List 头部元素后，用 llen，返回结果为 1。

List 结构可以用在多个场合，比如消息服务，通过 rpush 追加消息，其他客户端可以通过 lpop 或者 rpop 读取 List 的消息。

再比如网站的新闻列表，记录可以通过 lpush 放到新闻列表中，然后访问用户可以通过 lrange 0 9 来读取最新的 10 条记录。

> 对于消息服务需求，可使用 lpop 或者 rpop，如果列表为空，会返回一个 nil，导致消息订阅者不断尝试调用 pop 命令。另一种方式是 Redis 提供了带阻塞的（Block）的 pop 命令，blpop 或者 brpop。这两个命令会在 List 为空的时候处于等待状态，直到列表有元素，或者指定的时间到期为止：
>
> ```
> 127.0.0.1:6379> blpop platform:history 0
> 1) "platform:history"
> 2) "2011-10-22"
> ```

> blpop 参数可接受同时获取的多个列表，因此返回的是一个两元素列表，第一个元素是 Key 值，第二个是获取的元素值。
>
> blpop 总是返回先有元素的列表，参数 0 表示永远等待。

### 12.2.5 Redis Hash

Reids Hash 类似 Java 的 HashMap，允许存放多个 Key-Value。Spring Boot 在 Spring Session 中即采用了 Hash 结构来存放用户的 Session 数据，以实现 Web 系统的水平扩展。

Hash 有以下指令：

- hset key field value，给指定的 Key 设置一个字段值，如果值已经存在，则覆盖。返回 0 表示失败，返回 1 表示成功。
- hget key filed，获取指定 Key 的 filed 字段的值，如果不存在，返回 nil。
- hexists key filed，判断指定的 Key 的 filed 字段是否存在，返回 1 表示存在，0 表示不存在。
- hkeys key，返回 Key 所指定的 hash 所有的字段名。
- hgetall key，返回所有的字段名和字段值。
- hdel key field [field]，删除多个字段。

假设系统的用户 session 会话保存到 Redis 中，Key 值的格式是 session:{sessionId}，以下的 sessionId 假设为“1xac”：

```
127.0.0.1:6379> hset session:1xac name xiandafu
(integer) 1
127.0.0.1:6379> hset session:1xac ip 127.0.0.1
(integer) 1
127.0.0.1:6379> hget session:1xac name
"xiandafu"
127.0.0.1:6379> hkeys session:1xac
1) "name"
2) "ip"
127.0.0.1:6379> hgetall session:1xac
1) "name"
2) "xiandafu"
3) "ip"
```

```
4) "127.0.0.1"
127.0.0.1:6379>
```

Hash 的字段支持递增计算，这跟原始类型操作一样，是原子操作。

- hincby key field value，对 Key 指定的 Hash 数据中的 field 的值进行计算，增加整型 value。
- hincbyfloat key field value，对 Key 指定的 Hash 数据中的 field 的值进行计算，增加浮点数 value。

```
127.0.0.1:6379> hset website access 0
(integer) 0
127.0.0.1:6379> hincrby website access 1
(integer) 1
127.0.0.1:6379> hget website access
"1"
```

## 12.2.6 Set

Set 与 Java 中的 Java.util.Set 类似，代表了元素不重复的集合，Redis 的 Set 除了元素添加删除操作，还包含了集合的并集、交集等功能，可以用于统计访问网站所有的 IP，或者统计网站作者共同的粉丝等应用，常用的命令有：

- sadd key member [member ..]，添加元素，比如 sadd ip 192.168.0.1，向 ip 集合添加一个字符串，值是 192.168.0.1。
- srem key member [member …]，删除元素。
- smember key，返回一个集合中的所有元素。
- sinter key1 key2，返回两个集合共同的元素，key1 和 key2 分别代表两个集合。
- sinterstore key1 key2 key3，取的 key1 和 key2 的交集，并存放到 key3 集合中。
- sunion key1 key2，返回一个合并后的集合。
- sunionstore key1 key2 key3，合并 key1 和 key2 集合，并存放到 key3 集合中。

当 Set 用于统计网站访问 IP 的时候，实例如下：

```
127.0.0.1:6379> sadd ip 192.168.0.1
(integer) 1
127.0.0.1:6379> sadd ip 192.168.0.2
(integer) 1
```

```
127.0.0.1:6379> sadd ip 192.168.0.1
(integer) 0
127.0.0.1:6379> smembers ip
1) "192.168.0.2"
2) "192.168.0.1"
```

Set 用于查看两人共同好友，示例如下：

```
127.0.0.1:6379> sadd friend:xiandafu  lucy joel
(integer) 2
127.0.0.1:6379> sadd friend:lucy xiandafu joel tom
(integer) 3
127.0.0.1:6379>    sinterstore    friend:lucy-xiandafu    friend:xiandafu
friend:lucy
(integer) 1
127.0.0.1:6379> smembers friend:lucy-xiandafu
1) "joel"
127.0.0.1:6379>
```

第一行添加 lucy 和 joel 两个人到 xiandafu 的朋友列表中，朋友列表的 key 以 friend 为命名空间。

第二行添加了 xiandafu、joel、tom 三人到 lucy 的朋友列表中，现在我们分别有 xiandafu 和 lucy 的朋友列表，通过 sinterstor 可以获取两个朋友列表的交集，并存放到 Key 是“friend:lucy-xiandafu”的 Set 集合中。

> Redis 还提供了对 Sorted Sets 排序的集合的各种操作命令，由于篇幅原因，就不在本章说明了。

## 12.2.7 Pub/Sub

Reids 除了 NoSQL 特性，还提供了简单的消息服务，支持 publish/subscribe。Redis 客户端可以订阅一个或者多个频道（Channel），这种行为被称为 subscribe。其他 Redis 客户端向这些 Channel 发送消息，称为 publish，订阅这些频道的客户端能接收到这些消息。

publish/subscribe 消息模式很好地解耦了消息发送者和消息接收者。消息订阅者不需要知道发送者，发送者也不需要知道消息接收者。发送者发送的消息将被所有的消息订阅者接收。

publish/subscribe 模式在 Spring Boot 中可以应用于事件通知，如配置文件更新、缓存更新

等。在 Spring Cache 一章，会采用 Redis 的 NoSQL 特性作为数据缓存，也会使用 Pub/Sub 特性来发出缓存更新事件，从而让 Spring Boot 应用更新缓存。

Redis 的 subscribe 可以订阅一个到多个频道，假设订阅了一个 news 频道：

```
127.0.0.1:6379> subscribe news
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news"
3) (integer) 1
```

subscribe 命令会返回一个数组结构数据，第一行是固定的"subscribe"字符串，第二行是订阅的频道名字，第三行是数字，表示该频道总共有多少个订阅者。

redis-cli 一旦使用 subscribe，将一直等待频道的消息并输出到屏幕，因此，如果你还想在 redis-cli 中使用其他命令，比如 publish 消息，则需要重新打开一个终端，运行 redis-cli，使用以下 publish 命令：

```
127.0.0.1:6379> publish news "hello"
(integer) 1
127.0.0.1:6379> publish news "world"
(integer) 1
```

publish 命令可以向终端发送一条消息，并返回一个整数，表示有多少订阅者收到此消息。

如果回到订阅者的终端，则可以看到以下输出：

```
127.0.0.1:6379> subscribe news
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news"
3) (integer) 1

1) "message"
2) "news"
3) "hello"
1) "message"
2) "news"
3) "woorld"
```

Redis 还提供了订阅指定的模式（pattern），使用 psubscribe 命令：

```
127.0.0.1:6379> psubscribe news.*
```

订阅所有以 news 开头的频道：

支持的模式（patterns）有：

- news.*，所有 news.开头的频道；
- news-?，订阅 news-1、news-2 频道；
- news[123]，订阅 news-1、news-2、news-3 频道。

## 12.3 Spring Boot 集成 Redis

在 Spring Boot 中，引入 spring-boot-starter-data-redis 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

还需要配置 application.properties，指定 Redis 服务器的 IP、端口和密码：

```
spring.redis.host=127.0.0.1
spring.redis.password=Redis!123
spring.redis.port=6379
# 最大连接数
spring.redis.pool.max-active=8
```

属性 max-active 指定了 Spring Boot 应用的最大连接数，0 表示无限制。

为了测试是否集成成功，可以使用内置的 StringRedisTemplate 来进行测试，完成一个简单的 Redis 操作：

```
@Controller
@RequestMapping("/strredis")
public class RedisStringCrontroller {

  @Autowired
```

```
  private StringRedisTemplate redisClient;

  @RequestMapping("/setget.html")
  public @ResponseBody String env(String para  ) throws Exception{
    redisClient.opsForValue().set("testenv", para);
    String str = redisClient.opsForValue().get("testenv");
    return str;
  }
}
```

StringRedisTemplate 是 Spring Boot 默认提供的 Redis 操作接口，适合 Key 和 Value 都是字符串的情况，这种形式对于 Redis 来说（相对于 Redis 支持的二进制 Key-Value），容易阅读。

Spring Boot 也可以采用 JDK 序列化的方式来序列化 Key 和 Value 的 RedisTemplate 类，这是一个通用类，其实现可以提供不同的序列化方式。下一节将使用 String 序列化方式，然后是 JDK 序列化方式，以及我们可以定义一个 JSON 序列化方式。

> Redis 允许 Key 和 Value 是任意二进制形式，笔者认为最好还是使用字符作为 Key-Value 的形式，这样容易通过 Redis 客户端查看和管理。JSON 方式也是一种不错的方式，可以将 Value 序列化成 JSON 字符串。

上述 Java 代码可用 Redis 的 set 操作完成：

```
127.0.0.1:6379> set testenv 123
```

打开 redis-cli，查询 testenv，可以查看 Redis 操作是否成功：

```
127.0.0.1:6379> keys te*
1) "testenv"
127.0.0.1:6379> get testenv
"123"
127.0.0.1:6379>
```

## 12.4 使用 StringRedisTemplate

StringRedisTemplate 是 Spring Boot 内置的操作 Redis 的 API，另外一个内置的 API 是 RedisTemplate，StringRedisTemplate 的 API 假定所有的数据类型都是字符类型，比如 Key 是字符串，Value 也是字符串，List、Hash 中的元素值也是字符串。对于常见的 Spring Boot 应用系

统来说，使用字符串足够了，而且能方便地通过客户端管理工具管理。

StringRedisTemplate 继承了 RedisTemplate，与 RedisTemplate 不同的是重新设置了序列化策略，使用 StringRedisSerializer 类来序列化 Key-Value，以及 List、Hash、Set 等数据结构中的元素。

### 12.4.1 opsFor

- opsForValue，用来设置普通的 Key-Value

```
redisClient.opsForValue().set("testenv", "para");
redisClient.opsForValue().get("testenv");
```

相当于：

```
127.0.0.1:6379> set testenv para
127.0.0.1:6379> get testenv
```

- opsForList，用来操作 List 结构

```
redisClient.opsForList().leftPush("platform:message","hello,xiandafu");
redisClient.opsForList().leftPush("platform:message","hello,spring boot");
```

相当于以下 redis-cli 操作：

```
127.0.0.1:6379> lpush platform:message hello,xiandafu
127.0.0.1:6379> lpush platform:message hello,spring boot
```

可以通过 lrange 来查看 platform:message：

```
127.0.0.1:6379> lrange  platform:message  0 -1
1) "hello,xiandafu"
2) "hello,spring boot"
```

opsForList 提供了 lefPush、leftPushAll、leftPop、rightPush、rightPushAll、rightPop 等操作，也提供了 range 操作与 size 操作，用于查看 List 的长度。

- opsForHash，用来操作 Hash 数据结构

```
redisClient.opsForHash().put("cache", key, value);
String str = (String)redisClient.opsForHash().get("cache", key);
```

相当于以下 redis-cli 操作：

```
127.0.0.1:6379> hset cache key value
(integer) 1
127.0.0.1:6379> hget cache key
"value"
127.0.0.1:6379>
```

其他方法还有 delete、size、hasKey、keys、increment、multiGet 等。

## 12.4.2 绑定 Key 的操作

opsForxxx 每次都要输入 Key 作为第一个参数，你也可以通过 RedisTemplate 提供的 boundXXXOps()来指定一个 Key，返回 BoundXXXOperations。这样在 BoundXXXOperations 上的操作就不需要提供 Key 作为参数：

```
BoundListOperations operations = redisClient.boundListOps("somekey");
operations.leftPush("a");
operations.leftPush("b");
return String.valueOf(operations.size());
```

相当于以下 Redis 操作：

```
127.0.0.1:6379> lpush somekey a
(integer) 1
127.0.0.1:6379> lpush somekey b
(integer) 2
```

Spring Boot 提供了以下 bound 操作：

| 方 法 名 | 返回对象 | 描 述 |
|---|---|---|
| boundValueOps | BoundValueOperations | value 相关操作，如 set、get、append、incr 等 |
| boundListOps | BoundListOperations | List 相关操作，如 lpush、rpsh、lpop、rpop 等 |
| boundHashOps | BoundHashOperations | Hash 相关操作，如 hset、hget、hkeys 等 |

续表

| 方　法　名 | 返回对象 | 描　　述 |
|---|---|---|
| boundSetOps | BoundSetOperations | Set 相关操作 |
| boundZSetOps | ZSetOperations | Sorted Set 相关操作 |
| boundGeoOps | BoundGeoOperations | Geo，地理信息相关操作 |

## 12.4.3 RedisConnection

Spring Boot 提供了 RedisConnection 抽象，用于低级别 API 操作 Redis，具体实现有 JRedis 或者 Lettuce。

由于 Redis 所有数据结构都是二进制的，Spring Boot 对 StringRedisTemplate 做了一定抽象，通过内置的序列化机制将字符串序列化成 byte。Spring Boot 也提供了 RedisTemplate，默认使用 Java 的序列化机制将 Redis 数据序列化成 byte。

RedisConnection 提供了低级别的 API 操作，用 byte 数组作为参数操作 Redis 服务器。

```
@RequestMapping("/connectionset.html")
public @ResponseBody String connectionSet (final String key,final String value) throws Exception{
    redisClient.execute(new RedisCallback(){
      public Object doInRedis(RedisConnection connection) throws DataAccessException {
        try {
          connection.set(key.getBytes(), value.getBytes());
        } catch (UnsupportedEncodingException e) {
          throw new RuntimeException(e);
        }
        return null;
      }
    });
    return "success";
}
```

在这个例子中，也可以直接将 RedisConnection 转为 StringRedisConnection：

```
((StringRedisConnection)connection).set(key, value);
```

## 12.4.4 Pub/Sub

RedisTemplate 支持 Pub/Sub 功能，调用 convertAndSend 方法可以发送一条消息：

```
redisClient.convertAndSend("news", "hello,world");
```

convertAndSend 方法用于向 channal 发送消息。第一个参数是 channal 的名称，第二个参数是消息体。redisClient 将消息体序列化成字节后发送到 Redis Server。

为了订阅频道 news 的消息，需要实现 MessageListener 的 onMessage 方法，代码如下：

```
class MyRedisChannelListener implements MessageListener{

  public void onMessage(Message message, byte[] pattern) {
    byte[] channal = message.getChannel();
    byte[]  bs = message.getBody();
    try {
      String content = new String(bs,"UTF-8");
      String p = new String(channal,"UTF-8");
      System.out.println("get "+content+" from "+p);
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    }
  }

}
```

MessageListener 接口同 Java 中的多数消息处理框架一样，有一个 onMessage 方法，参数说明如下：

- Message，可以通过调用 getBody 方法获取消息内容，返回的是字节数组。因为我们在发送消息时采用的是 StringRedisTemplate，也就是发送的时候，消息编码就是字符通过 UTF-8 转成字节，因此我们反过来直接使用 UTF-8 构造字符串即可。getChannel 用于得到消息所在的频道，要注意，Spring Boot 应用通常通过一个 MessageListener 来实现 Redis 消息监听，因此，应用有时候需要根据 channal 名字来区分。
- Pattern，订阅模式。

编写好 MessageListener 后，还需要添加一些固定的 Java 代码来设置监听器：

```
@Bean
MessageListenerAdapter listenerAdapter( ) {
  return new MessageListenerAdapter(new MyRedisChannelListener());
}

@Bean
RedisMessageListenerContainer container(RedisConnectionFactory
connectionFactory,
                                        MessageListenerAdapter listenerAdapter) {
  RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
  container.setConnectionFactory(connectionFactory);
  container.addMessageListener(listenerAdapter, new
PatternTopic("news.*"));
  return container;
}
```

MessageListenerAdapter 主要用来对消息进行序列化工作，默认采用 StringRedisSerializer，我们将会在下一节序列化策略中讲述。

RedisMessageListenerContainer 的主要作用是在 Redis 客户端接收到消息后，通过 PatternTopic 派发消息到对应的消息监听者，并构造一个线程池任务来调用 MessageListener。

## 12.5 序列化策略

前面提过，Spring Boot 默认提供了 StringRedisTemplate 和 RedisTemplate，前者用于操作包含字符串的数据结构，后者则使用了 JDK 的序列化策略。

事实上 StringRedisTemplate 继承了 RedisTemplate，设置了不同的序列化策略：

```
public class StringRedisTemplate extends RedisTemplate<String, String> {
  public StringRedisTemplate() {
    RedisSerializer<String> stringSerializer = new StringRedisSerializer();
    setKeySerializer(stringSerializer);
    setValueSerializer(stringSerializer);
    setHashKeySerializer(stringSerializer);
    setHashValueSerializer(stringSerializer);
  }
}
```

RedisTemplate 则采用默认的序列化策略 JdkSerializationRedisSerializer，这两种序列化策略都实现了 RedisSerializer 接口，定义如下：

```
public interface RedisSerializer<T> {
  // 序列化Java对象为字节数组
  byte[] serialize(T t) throws SerializationException;
  // 反序列化字节数组到Java对象
  T deserialize(byte[] bytes) throws SerializationException;
}
```

serialize 方法将默认的对象转为字节数组，以提供给 RedisConnection 操作，deserialize 则将 RedisConnection 读取的 byte[]转为对象。

以下是 StringRedisSerializer 的实现，默认使用 UTF-8 编码，将字符串转为字节数组，或者相反过程：

```
public class StringRedisSerializer implements RedisSerializer<String> {

  private final Charset charset;

  public StringRedisSerializer() {
    this(Charset.forName("UTF8"));
  }

  public StringRedisSerializer(Charset charset) {
    Assert.notNull(charset, "Charset must not be null!");
    this.charset = charset;
  }

  public String deserialize(byte[] bytes) {
    return (bytes == null ? null : new String(bytes, charset));
  }

  public byte[] serialize(String string) {
    return (string == null ? null : string.getBytes(charset));
  }
}
```

## 12.5.1 默认序列化策略

Spring Session 和 Spring Cache 都可以使用 Redis 作为缓存服务器，并使用默认的序列化策略，即 JdkSerializationRedisSerializer。

以下例子将使用 get 和 set 操作保存一个字符串到 Redis 中，以及保存一个 User 示例到 Redis 中。

首先，引入 RedisTemplate：

```
@Autowired
@Qualifier("redisTemplate")
private RedisTemplate redisClient;
```

必须使用@Qualifier("redisTemplate")，否则 Spring Boot 会误认为有可能是 StringRedisTemplate 而报错。

创建一个可序列的 User 对象：

```
public static class User implements java.io.Serializable{
  int id ;
  String name ;
  Date date = new Date();
  // 忽略 getter 和 setter
  public static User getSampleUser(){
    User user = new User();
    user.setId(123);
    user.setName("abc");
    return user;
  }
}
```

User 对象实现了 java.io.Serializable 接口，从而保证能被 RedisTemplate 序列化存储到 Redis 中。测试代码如下：

```
redisClient.opsForValue().set("key-0","hello");
redisClient.opsForValue().set("key-1", User.getSampleUser());
```

如果使用 redis-cli 查询 key-0，会得到以下错误：

```
127.0.0.1:6379> get key-0
(nil)
```

实际上，“key-0”是以 Java 序列化形式存储的，通过以下命令找到真正的 Key 值：

```
127.0.0.1:6379> keys *key-0
1) "\xac\xed\x00\x05t\x00\x05key-0"
127.0.0.1:6379> get "\xac\xed\x00\x05t\x00\x05key-0"
"\xac\xed\x00\x05t\x00\x05hello"
```

JDK 序列化对象可以简单地理解为第一部分是对象的描述信息，在这个例子中是“\xac\xed\x00\x05t\x00\x05”，第二部分是对象的数据内容，正是我们要的“hello”。

我们可以在 Java 中通过 opsForValue().get 方法来获取放入 Redis 中的值：

```
String value = (String)redisClient.opsForValue().get("key-0");
User uer = (User)redisClient.opsForValue().get("key-1");
```

## 12.5.2 自定义序列化策略

使用 RedisTemplate 提供的默认序列化策略不是很方便，我们可以自己指定序列化测试，比如对于默认的 RedisTemplate，我们可以指定 Key 的序列化策略为 StringRedisSerializer。

创建一个配置类：

```
@Configuration
public class RedisConfig {

  @Bean("strKeyRedisTemplate")
  public RedisTemplate<Object, Object> strKeyRedisTemplate(
    RedisConnectionFactory redisConnectionFactory)
    throws UnknownHostException {
    RedisTemplate<Object, Object> template = new RedisTemplate<Object, Object>();
    template.setConnectionFactory(redisConnectionFactory);
    RedisSerializer<String> stringSerializer = new StringRedisSerializer();
    template.setKeySerializer(stringSerializer);
    return template;
  }
}
```

这里创建了一个名为“strKeyRedisTemplate”的 RedisTemplate 实例。以下代码对 Key 的序列化策略进行了重新设定：

```
RedisSerializer<String> stringSerializer = new StringRedisSerializer();
template.setKeySerializer(stringSerializer);
```

因此，如果我们用这个 RedisTemplate 来操作 Redis，则能看到 Key 值正是我们想要的字符串形式，而不是二进制序列化的形式：

```
127.0.0.1:6379> get key-0
"\xac\xed\x00\x05t\x00\x05hello"
127.0.0.1:6379>
```

Spring Boot 除了提供这两种序列化方式，还提供了 JSON 的序列化方式，以下代码是使用 Jackson 作为默认的序列化方式：

```
@Bean("jsonRedisTemplate")
public RedisTemplate<Object, Object> redisTemplate(
  RedisConnectionFactory redisConnectionFactory)
  throws UnknownHostException {
  RedisTemplate<Object, Object> template = new RedisTemplate<Object, Object>();
  template.setConnectionFactory(redisConnectionFactory);
  template.setDefaultSerializer(new GenericJackson2JsonRedisSerializer());

  return template;
}
```

GenericJackson2JsonRedisSerializer 包含了一个默认的 ObjectMapper，也支持通过构造函数传入一个系统定义好的 ObjectMapper。比如 ObjectMapper 已经是 Spring Boot 定义好的（参考第 4 章）：

```
@Bean("jsonRedisTemplate")
public RedisTemplate<Object, Object> redisTemplate(
  RedisConnectionFactory redisConnectionFactory,ObjectMapper mapper)
  throws UnknownHostException {
  // 省略，代码同上
  template.setDefaultSerializer(new GenericJackson2JsonRedisSerializer(mapper));
```

```
  return template;
}
```

在 12.4.4 节中，使用的 MessageListenerAdapter 默认序列化策略是 StringRedisSerializer，如果 Pub 消息采用的序列化策略不是 StringRedisSerializer，可以改成相应的序列化策略：

```
@Bean
MessageListenerAdapter listenerAdapter( ) {
  MessageListenerAdapter adapter = new MessageListenerAdapter(new
MyRedisChannelListener());
  // 改成使用 JDK 序列化机制
  adapter.setSerializer(new JdkSerializationRedisSerializer());
  return adapter;
}
```

# 第 13 章 Elasticsearch

## 13.1 Elasticsearch 介绍

Elasticsearch，简称 ES。是一个全文搜索服务器，也可以作为 NoSQL 数据库，存储任意格式的文档和数据，同时，也可以做大数据的分析，是一个跨界开源产品。ES 有如下特点：

- 全文搜索引擎，ES 是建立在 Lucene 上的开源搜索引擎，可以用来进行全文搜索、地理信息搜索。Wikipedia、GitHub、StackOverFlow 等网站均使用 ES。
- 文档存储和查询，可以像 NoSQL 那样存储任意格式文档，并能根据条件查询文档。
- 大数据分析，ES 号称能准确实时地进行大数据分析，数据量从 TB 到 PB，国内外很多大公司都用 ES 做大数据分析。
- ES 提供了 REST API，用来简化对 ES 的操作。因此可以使用任何语言的客户端，同时也提供 Java API，Spring Boot 也对 REST API 进行了封装，简化了开发。
- ES 常常配合传统数据库一起使用，ES 用来负责大数据的查询、搜索、统计分析。

### 13.1.1 安装 Elasticsearch

需要 JDK 环境，首先要确保为 JDK8 以上版本。

```
java -version
```

- 访问地址 www.elastic.co/downloads，下载最新的 ES 版本，选择 Elasticsearch 进行下载。
- 直接解压 elasticsearch-5.4.3.zip 到某个目录。
- 进入 bin 目录，运行./elasticsearch，Linux 下不能使用 root 直接运行，最好为运行 ES 创建一个新的用户。
- 看到以下提示，表示安装成功：

```
[CIZ4b2B] publish_address {127.0.0.1:9300}, bound_addresses {[fe80::1]:9300}, {[::1]:9300}, {127.0.0.1:9300}

publish_address {127.0.0.1:9200}, bound_addresses {[fe80::1]:9200}, {[::1]:9200}, {127.0.0.1:9200}
```

9200 端口是对外的 RESTFul 接口，9300 端口是 ES 内部使用的端口。

- 打开浏览器，访问以下地址：

```
http://localhost:9200/
```

浏览器有以下输出：

```
{
  "name" : "hGl0OvN",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "hNy6F-8CRtCnNL5D3sAH8Q",
  "version" : {
    "number" : "5.4.3",
    "build_hash" : "eed30a8",
    "build_date" : "2017-06-22T00:34:03.743Z",
    "build_snapshot" : false,
    "lucene_version" : "6.5.1"
  },
  "tagline" : "You Know, for Search"
}
```

以上输出比较简单，name 代表了这台 ES 的名字，可以配置指定的名字，另外一个是 cluster_name，默认名字是 elasticsearch。ES 的集群方式是通过广播在同一个网络中寻找

cluster_name 一样的 ES，cluster_name 对应的名字是同一个集群的标记。

出于安全原因，默认是只允许本机访问，如果想让特定机器能访问 ES，需要配置 config/elasticsearch.yml，增加 network.host 配置，以下是允许所有机器访问 ES：

```
network.host: 0.0.0.0
```

> **注意：** 默认的 ES 对中文搜索不友好，需要安装额外的插件，本章使用了官方推荐的 smartcn 插件。
>
> - 进入 bin 目录，运行./elasticsearch-plugin install analysis-smartcn。
> - 如果想使用别的分词插件，可使用./elasticsearch-plugin remove analysis-smartcn。
> - 最后需要重启所有的 ES 节点。

### 13.1.2 Elasticsearch 的基本概念

ES 有一些基本概念，掌握这些基本概念对理解 ES 有很大帮助。

- Index，Index 是文档（Document）的集合，Index 下面包含了 Type，用于对 Document 进一步分类。可以理解为 ES 中的 Index 相当于数据库，而 Type 相当于数据库中的表，ES 中可以轻易地联合 Index 和 Type 来搜索数据，数据库却不能。
- Type，用来进一步组织 Document，一个 Index 下可以有多个 Type，比如用户信息是一个 Type，用户的支付记录是一个 Type。
- Document，文档是 ES 能够存储和搜索的基本信息，类似数据库表行数据，Document 为 JSON 格式，文档属于 Type。
- Node（节点），节点是集群里的一台 ES Server，用于文档的存储和查询。应用可以只有一个节点，也可以由上百个节点组成集群来存储和搜索数据。每个节点都有一个节点名字，以及所属集群的名字。
- 集群，同样集群名的节点将组合为 ES 集群，用来联合完成数据的存储和搜索。默认的集群名字是 elasticsearch。
- 分区（Shards）和复制（Replicas），每个 Index 理论上都可以包含大量的数据，超过了单个节点的存储限制，而且，单个节点处理那么大的数据，将明显限制存储和搜索性能。为了解决这个问题，ES 会进一步将 Index 在物理上细分为多个分区，而且这些分区会按照配置复制到多个节点，Index 的分区称为主分区，复制的分区称为复制分区。这样的好处是既保证数据不会丢失，又提高了查询的性能。

每个分区是一个独立的工作单元，可以完成存储和搜索功能，每个分区能存储最多 2147483519 个文档。

> **注意：**本章将以对商品的搜索和分析作为例子。创建的 Index 为 product，并创建 Type，名字为 book，book 文档有名称、出版日期、描述等属性，以下是《北京 100 种小吃》的属性：
>
> ```
> {
>  "name" : "北京 100 种小吃",
>  "type":"food",
>  "postDate" : "2009-11-15T14:12:12",
>  "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚等"
> }
> ```

## 13.2 使用 REST 访问 Elasticsearch

Elasticsearch 之所以受欢迎的一个重要原因是其基于 RESTFul 接口，使用起来简单方便，ES 操作基本上分为以下几类：

- 文档的增删改查；
- 全文搜索；
- 聚合搜索；
- 处理人类语言；
- 地理位置搜索。

由于篇幅有限，本书将重点介绍前三种操作，不会涉及所有 Elasticsearch 的知识。

### 13.2.1 添加文档

使用 PUT 添加文档，采用 curl，在命令行输入：

```
curl -XPOST 'localhost:9200/product/book/1?pretty' -H 'Content-Type:
application/json' -d'
{
   "name" : "北京 100 种小吃",
   "type":"food",
   "postDate" : "2009-11-15",
```

```
    "message" : "介绍了北京小吃，如炸酱面、卤煮、驴打滚等"
}
'
```

product 表示 Index，book 表示 Type，数字 1 是文档的主键，主键可以是任意形式，如果未指定主键，ES 会自动生成一个唯一主键，pretty 是可选的，ES 输出的时候会格式化输出结果，更加美观。

> 本章的所有 REST 操作都通过 curl 命令完成，如果你还不熟悉 curl，请参考第 3 章关于 curl 介绍，或者使用 Postman 工具。

postDate 的格式是 ES 默认的日期格式之一，为 yyyy-MM-dd，ES 还默认了多种格式为日期格式，遇到这些格式，ES 会自动认为类型为日期类型，以下三种格式数据 ES 都会处理成日期类型。

- yyyy-MM-dd，如 2009-11-15；
- yyyy-MM-dd'T'HH:mm:ss，如 2009-11-15T13:14:21；
- yyyy-MM-dd'T'HH:mm:ss.SSS，如 2009-11-15T13:14:21.389。

操作后的响应是：

```
{
  "_index" : "product",
  "_type" : "book",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : true
}
```

字段_id 表示该文档的主键，如果在添加文档的时候并未指定主键，系统默认生成一个唯一主键。

_shards 表示分区信息，total 为 2 表示有两个分区（包括主分区），successful 为 1 表示成

功复制了一份。

_version 代表了文档版本号，每一次修改都会递增，注意 ES 并不会存储文档修改的各个版本。

## 13.2.2 根据主键查询

根据主键查询：

```
curl -XGET 'localhost:9200/product/book/1?pretty'
```

控制台输出：

```
{
  "_index" : "product",
  "_type" : "book",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "name" : "北京100种小吃",
    "type" : "food",
    "postDate" : "2009-11-15T14:12:12",
    "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚等"
  }
}
```

_source 表示查询的文档，这正是我们存入的文档。如果你只想看到 source 部分，可以加一个_souorce：

```
curl -XGET 'localhost:9200/product/book/1/_source?pretty'
```

## 13.2.3 根据主键更新

根据主键更新同新增一样，需要指定主键，然后更新整个文档。

```
curl -XPUT 'localhost:9200/product/book/1?pretty' -H 'Content-Type:
application/json' -d'
```

```
{
   "name" : "北京 108 种小吃",
   "type":"food",
   "postDate" : "2009-11-15T14:12:12",
   "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚等"
}
'
```

控制台输出：

```
{
  "_index" : "product",
  "_type" : "book",
  "_id" : "1",
  "_version" : 2,
  "result":"updated",
  "_shards" : {
   "total" : 2,
   "successful" : 1,
   "failed" : 0
  },
  "created" : false
}
```

修改文档同添加文档类似，不同的地方是_version 递增了，result 的值是 updated，表示更新成功。

如果只想局部更新，可以采用 POST，使用_update，比如只更新 message 字段：

```
curl -XPOST 'localhost:9200/product/book/1/_update?pretty' -H 'Content-Type:
application/json' -d'
{
  "doc":{
    "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚，还有胶圈等!!!"
  }
}
'
```

字段 doc 包含了要更新的文档的片段。

如果并发修改文档，可以使用 version 字段实现乐观锁，如果修改的文档的 version 和传入的 version 不一致，则修改失败。

```
curl -XPOST 'localhost:9200/product/book/1/_update?pretty&version=2' -H
'Content-Type: application/json' -d'
  {
   "doc":{
     "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚，还有焦圈等!!!"
    }
  }
'
```

如果当前文档版本号不是 2 的话，则修改失败，控制台抛出如下错误：

```
{
  "error" : {
    "root_cause" : [
      {
        "type" : "version_conflict_engine_exception",
        ......
      }
    ],
    "type" : "version_conflict_engine_exception",
    ......
  },
  "status" : 409
}
```

可以看到 status 为 409，type 字段为 version_conflict_engine_exception，表示版本冲突，更新失败。

> 在 HTTP 中，状态码 409 表示资源冲突。

## 13.2.4 根据主键删除

使用 DELETE 删除指定主键的文档：

```
curl -XDELETE 'localhost:9200/product/book/1?pretty'
```

控制台输出：

```
{
  "found" : true,
  "_index" : "product",
  "_type" : "book",
  "_id" : "1",
  "_version" : 10,
  "result" : "deleted",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  }
}
```

如果删除不存在的文档，found 值为 false：

```
{
  "found" : true,
  ....
}
```

### 13.2.5 搜索文档

ES 提供了强大的搜索功能，搜索参数可以在 url 后面，也可以放到 body 中。使用 GET 方法：

```
curl -G --data-urlencode 'q=message:驴打滚'
'localhost:9200/product/book/ _search?pretty'
```

或者更通用的 POST 方法：

```
curl -XPOST 'localhost:9200/product/book/_search?pretty' -H 'Content-Type:
application/json' -d'
{
  "query" : {
```

```
        "match": { "message" : "驴打滚"}
    }
}
'
```

> **注意**：因为关键字里包含了中文，需要 curl 进行 URI 编码，所以才使用了--data-urlencode 'q=message:驴打滚'，参数 "-G" 表示这是一个 GET 请求，如果不加 "-G"，curl 默认发出 POST 请求，导致 ES 返回一个 406 不支持 POST 请求错误响应。

无论哪种查询，都会有大概类似如下的响应：

```
{
  "took" : 7,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.789034,
    "hits" : [
      {
        "_index" : "product",
        "_type" : "book",
        "_id" : "2",
        "_score" : 0.789034,
        "_source" : {
          "name" : "北京 100 种小吃",
          "type" : "food",
          "postDate" : "2009-11-15",
          "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚等"
        }
      },
      {
        "_index" : "product",
```

```
        "_type" : "book",
        "_id" : "1",
        "_score" : 0.77069074,
        "_source" : {
          "name" : "北京100种小吃",
          "type" : "food",
          "postDate" : "2009-11-15",
          "message" : "介绍了北京小吃，如炸酱面，卤煮，驴打滚，还有焦圈等123!!!"
        }
      }
    ]
  }
}
```

hits 包含了查询结果，在本例中，只有 2 条，Index 是 product，Type 是 book，主键是 1 和 2，_score 是搜索引擎概念，表示查询相关度，分数越高，表示此文档与关键字期望的结果的匹配程度高。

除了全文搜索，也可精确搜索，使用 term 进行精确搜索：

```
curl -XGET 'localhost:9200/product/book/_search?pretty' -H 'Content-Type: application/json' -d'
{
  "query" : {
    "term": { "type" : "food"}
  }
}
'
```

如果需要完成翻页功能，可以使用 from 和 size：

```
curl -XGET 'localhost:9200/product/book/_search?pretty' -H 'Content-Type: application/json' -d'
{
  "from":0,"size":5,
  "query" : {
    "term": { "type" : "food"}
  }
```

```
}
'
```

如果需要知道查询总数，则使用_count 代替_search。

查询书中类型为菜谱的书的总数：

```
curl -XGET 'localhost:9200/product/book/_count' -H 'Content-Type:
application/json' -d'
{
    "query" : {
        "term": { "type" : "food"}
    }
}
'
```

如果要联合条件查询，则可以使用 must 关键字：

```
curl -XGET 'localhost:9200/product/book/_search?pretty' -H 'Content-Type:
application/json' -d'
{
  "from":0,"size":5,
  "query" : {
    "bool": {
      "must": { "match": { "message" : "驴打滚"}},
      "must": { "term": { "type" : "IT"}}

    }
  }
}
'
```

> **注意：** ES 具有强大的搜索功能，限于篇幅，将不做进一步介绍，关于排序、分组聚合、统计分析、地理位置查询等功能请参考 ES 官网文档 https://www.elastic.co/。

## 13.2.6 联合多个索引搜索

考虑到 ES 中每个 Index 存储的文档数量仍然有限，在将文档放入 Index 中的时候，可以增

加日期后缀，比如 product、product2008、product2009 这三个 Index。ES 在搜索的时候，可以引用多个索引。

- /_search，搜索所有索引的所有类型。
- _all/book，查询所有索引类型为 book 的文档。
- product*/book，查询以 product 开头的所有索引。
- product2008,product2009/book，搜索 2008 年和 2009 年的索引。
- product/book,pc，搜索索引为 product，类型为 book 和 pc 的。
- _all/book,pc，搜索所有类型为 book 和 pc 的。

## 13.3 使用 RestTemplate 访问 ES

ES 是 RESTFul 接口，因此直接构造 REST 请求和解析 REST 返回的 JSON 即可访问 ES 服务器，本节使用 Spring Boot 提供的 RestTemplate 来编写 Java 代码。

### 13.3.1 创建 Book

我们知道 ES 查询将返回 JSON 格式，如果查询有返回值，source 属性则包含了返回结果，我们首先创建一个 Book 类，用来反序列化 ES 返回的查询结果，代码如下：

```
package com.bee.sample.ch15.entity;

import java.util.Date;

public class Book {
    String name;
    String message;
    Date postDate;
    // 忽略 getter setter 方法
}
```

### 13.3.2 使用 RestTemplate 获取搜索结果

为了测试程序，我们创建一个 RestController 来测试我们的方法，新建 RestClientController：

```
@RestController
public class RestClientController {

  @RequestMapping("/restclient/book/{id}")
  public String getLogById(@PathVariable String id) throws Exception {
    Book book = null;
    ....
    return log.getMessage();
  }
}
```

程序期望当访问 Spring Boot 应用，地址是 http://localhost:8080/book/1 的时候，返回该 id 对应的书籍信息。

本节使用 Spring Boot 的 RestTemplate，关于 RestTemplate 更多的介绍，请参考第 10 章关于 REST 的内容。

```
Book book = null;
RestTemplate template = new RestTemplate();
Map<String,Object> paras = new HashMap<>();
paras.put("id", id);
// 得到响应的 JSON
String str = template.getForObject("http://172.16.86.56:9200/product/book/
{id}", String.class,paras);

ObjectMapper mapper = new ObjectMapper();
JsonFactory factory = mapper.getFactory();
JsonParser parser = factory.createParser(str);
// 只对返回的 JSON 的 source 字段感兴趣
JsonNode root = mapper.readTree(parser);
JsonNode sourceNode = root.get("_source");
// 将 source 字段的文档部分映射到 Book 对象
book  = mapper.convertValue(sourceNode, Book.class);
return book.getMessage();
```

本例使用 Jackson 来解析 ES 返回的存在_source 节点的数据。

- 首先通过 RestTemplate，按照 ES 的格式，请求 id 为 1 的书籍信息，并返回字符串。

- 使用 Jackson 获取到_source 节点。从 13.2.2 节知道，ES 返回的信息存放在_source 节点中。
- 反序列化 sourceNode 到 Book 对象。

> 如果你熟悉 ES 的 REST 接口，可以用 RestTemplate 来处理 ES 的任何查询，而不需要后面讲的 Spring Data Elastic，不过 Spring Data 使得开发更为简单，容易维护。
>
> Spring Data 的缺点是不支持复杂的查询和统计分析等，最好使用 ES 的 REST 方式来完成，另外 ES 的版本演进跟其他 NoSQL 一样，特别快，新的功能 Spring Data 也暂时无法支持。在本书写作的 5 个月时间内，ES 版本先后经历了 5.2.2、5.4.3。当最后交稿的时候，版本已经是 5.5.2 了。

## 13.4 Spring Data Elastic

Spring Data Elastic 是 Spring 官方提供的访问 ES 的方式，相对于直接 REST 访问，它有以下优势：

- 完善的封装，Spring Data Elastic 遵循 Spring Data 规范，你只要会使用 Spring Data，比如 Spring JPA，就能使用 Elastic Data。
- 屏蔽了 ES REST 接口的复杂性，就像调用普通方法那样调用 Elastic Data，Spring 会自动调用 Elastic 底层 API 来完成查询。

### 13.4.1 安装 Spring Data

对于 Spring Boot，需要添加以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

Spring Boot 的 application.properties 配置 ES 的访问地址：

```
spring.data.elasticsearch.cluster-nodes=127.0.0.1:9300
```

## 13.4.2 编写 Entity

同 Spring Data 其他技术一样，首先需要完成一个 Entity：

```
import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;
@Document(indexName="product" ,type="book")
public class BookEntity {
    @Id
    String id;
    String name;
    String message;
    Date postDate;
    String type ;
    // 忽略 getter 和 setter 方法
}
```

@Document 表示这是一个 Elastic Data，indexName 和 type 对应于 Elasticsearch 的 Index 和 Type。本例中的索引是 product，类型是 book。

@Id 声明了文档的主键，同 Spring Data 一致。

## 13.4.3 编写 Dao

```
import org.springframework.data.repository.CrudRepository;

import com.bee.sample.ch15.entity.BookEntity;

public interface BookDao extends CrudRepository<BookEntity, String> {
    public List<BookEntity> getByMessage(String key);
}
```

Dao 继承了 Spring Data 的 CrudRepository，因此自带了简单的增删改查操作，我们再添加一个 getByMessage 方法，这是一个标准 Spring Data 命名，意味着 Elastic Data 会查询 message 字段，

如果是 getByMessageAndType (String key,String type)，则 Elastic Data 会联合查询 message 和 type 两个字段。

以下是 Spring Data Elastic 命名与 ES REST 接口的对应关系。

| 关 键 字 | 例 子 | ES REST |
|---|---|---|
| And | findByNameAndPrice | {"bool" : {"must" : [ {"field" : {"name" : "?"}}, {"field" : {"price" : "?"}} ]}} |
| Or | findByNameOrPrice | {"bool" : {"should" : [ {"field" : {"name" : "?"}}, {"field" : {"price" : "?"}} ]}} |
| Is | findByName | {"bool" : {"must" : {"field" : {"name" : "?"}}}} |
| Not | findByNameNot | {"bool" : {"must_not" : {"field" : {"name" : "?"}}}} |
| Between | findByPriceBetween | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : ?, "include_lower" : true,"include_upper" : true}}}}} |
| LessThanEqual | findByPriceLessThan | {"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?, "include_lower" : true,"include_upper" : true}}}}} |
| GreaterThanEqual | findByPriceGreaterThan | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null, "include_lower" : true,"include_upper" : true}}}}} |
| Before | findByPriceBefore | {"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?, "include_lower" : true,"include_upper" : true}}}}} |
| After | findByPriceAfter | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null, "include_lower" : true,"include_upper" : true}}}}} |
| Like | findByNameLike | {"bool" : {"must" : {"field": {"name" : {"query" : "?*","analyze_wildcard" : true}}}}} |
| Containing | findByNameContaining | {"bool" : {"must" : {"field": {"name" : {"query" : "?","analyze_wildcard" : true}}}}} |
| In | findByNameIn(Collection names) | {"bool" : {"must" : {"bool": {"should" : [ {"field" : {"name" : "?"}}, {"field" : {"name" : "?"}} ]}}}} |
| True | findByAvailableTrue | {"bool" : {"must" : {"field": {"available" : true}}}} |

续表

| 关 键 字 | 例 子 | ES REST |
|---|---|---|
| False | findByAvailableFalse | {"bool" : {"must" : {"field": {"available" : false}}}} |
| OrderBy | findByAvailableTrueOrderByNameDesc | {"sort" : [{ "name" : {"order" : "desc"} }],"bool" : {"must" : {"field" : {"available" : true}}}} |

如果查询包含翻页，则可以使用 Pageable 对象：

```
public Page<BookEntity> getByMessage(String key, Pageable pageable);
```

> 可以参考第 6 章关于 Spring Data 的内容。

### 13.4.4 编写 Controller

我们编写一个 ElasticDataController 来测试 BookDao，第一个是测试内置的 findById，第二个是测试 getByMessage 方法，第三个是测试搜索和翻页功能。

```
@RestController
public class ElasticDataController {

  @Autowired
  BookDao dao;
  @RequestMapping("/springdata/book/{id}")
  public String getBookById(@PathVariable String id) {
    // 测试内置的 findById
    Optional<BookEntity> opt = dao.findById(id);
    BookEntity book = opt.get();
    return book;
  }

  @RequestMapping("/springdata/search/{key}")
  public List<BookEntity> search(@PathVariable String key) {
    // 测试全文搜索
    List<BookEntity> list = dao.getByMessage(key);
    return list;
```

```
    }

    @RequestMapping("/springdata/search/{key}/{page}")
    public  List<BookEntity>   search(@PathVariable  int  page,@PathVariable
String key)  {
      int numberOfPage = 5;
      PageRequest request  =PageRequest.of(page, numberOfPage);
      // 全文搜索翻页
      Page<BookEntity> pages = dao.getByMessage(key,request);
      long total = pages.getTotalElements();
      long totalPage = pages.getTotalPages();
      List<BookEntity> list = pages.getContent();
      return list;
    }
```

通过这个例子，我们能看到 Spring Data Elastic 屏蔽了 REST 访问 ES 的复杂性，初学者不需要了解 ES 接口规范就能使用 ES。不过，便利性对开发人员来说也是把双刃剑，会忽略底层的一些实现细节和 ES 的接口规范，知识了解得不会太深入。

# 第 14 章 Cache

本章介绍 Spring Boot 应用系统中 Cache 的一般概念，Spring Cache 对 Cache 进行了抽象，提供了@Cacheable、@CachePut、@CacheEvict 等注解。Spring Boot 应用基于 Spring Cache，既提供了基于内存实现的缓存管理器，可以用于单体应用系统，也集成了 Redis、EhCache 等缓存服务器，可用于大型系统或者分布式系统。

Spring Boot 使得应用系统能很方便地使用缓存，极大提高了应用系统的性能和吞吐量。

本章最后深入剖析 Spring Boot 提供的 RedisCacheManager 代码，并扩展出一个性能更好的一二级缓存系统原型。

## 14.1 关于 Cache

应用系统需要通过 Cache 来缓存不经常改变的数据以提高系统性能和增加系统吞吐量，避免直接访问数据库等低速的存储系统。缓存的数据通常存放在访问速度更快的内存中或者是低延迟存取的存储器、服务器上。应用系统缓存通常有以下作用：

- 缓存 Web 系统的输出，如伪静态页面。
- 缓存系统中不经常改变的业务数据，如用户权限、字典数据、配置信息等。

本章主要介绍通过 Spring Boot 来缓存应用系统中不常改变的数据，并以缓存菜单项来举列子。

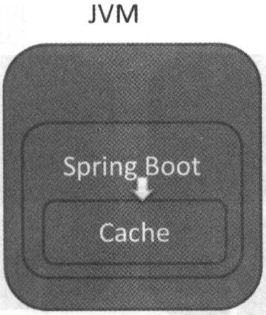
### 14.1.1 Cache 的组件和概念

Cache 相关的组件与概念如下：

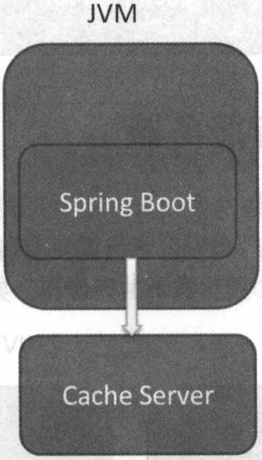
- CacheManager，用来创建、管理、管理多个命名唯一的 Cache，如组织机构缓存、菜单项的缓存、菜单树的缓存等。
- Cache 类似 Map 那样的 Key-Value 存储结构，Value 部分通常包含了缓存的对象，通过 Key 来取得缓存对象。
- 缓存项，存放在缓存里的对象，常常需要实现序列化接口，以支持分布式缓存。
- Cache 存储方式，缓存组件可以将对象放到内存或其他缓存服务器上，Spring Boot 提供了一个基于 ConcurrentMap 的缓存，同时也集成了 Redis、EhCache 2.x、JCache 缓存服务器等。
- 缓存策略，通常 Cache 还可以有不同的缓存策略，如设置缓存最大的容量，缓存项的过期时间等。
- 分布式缓存，缓存通常按照缓存数据类型存放在不同缓存服务器上，或者同一类型的缓存按照某种算法、不同 Key 的数据存放在不同的缓存服务器上。
- Cache Hit，从 Cache 中取得期望的缓存项，我们通常称之为缓存命中。如果没有命中则称之为 Cache Miss，意味着需要从数据来源处重新取出并放回 Cache 中。
- Cache Miss，缓存丢失，根据 Key 没有从缓存中找到对应的缓存项。
- Cache Evication，缓存清除操作。
- Hot Data，热点数据，缓存系统能调整算法或者内部存储方式，使得最有可能频繁访问的数据能被尽快访问到。
- On-Heap，Java 分配对象都是在堆内存中，有最快的获取速度。由于虚拟机的垃圾回收管理机制，缓存放入过多的对象会导致垃圾回收时间过长，从而有可能影响性能。
- Off-Heap，堆外内存，对象存放在虚拟机分配的堆外内存中，因此不受垃圾回收管理机制的管理，不影响系统性能，但堆外内存的对象要被使用，还要序列化成堆内对象。很多缓存工具会把不常用的对象放到堆外，把热点数据放到堆内。

### 14.1.2 Cache 的单体应用

单体应用中，Cache 可以与应用系统位于一个虚拟机中，这样的好处是访问速度最快。如果缓存数量大，可以通过一定的策略删除访问较少的缓存项，比如 Ehcache 就提供了删除访问较少数据，以及将较少访问的数据暂存在堆外或者硬盘上的功能，如下图所示。

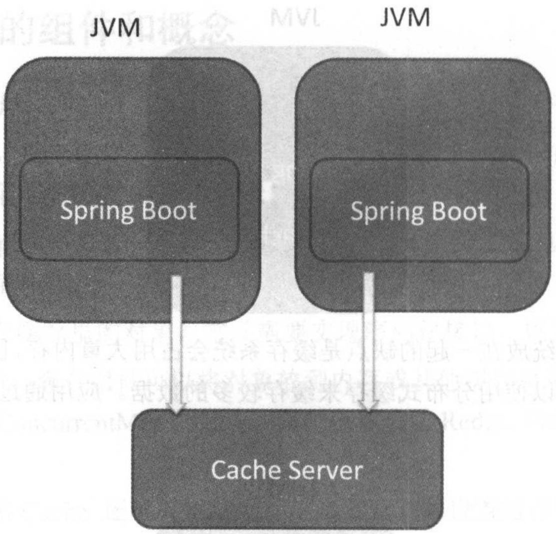



单体应用与缓存系统放在一起的缺点是缓存系统会占用大量内存，因此，对于有些缓存项，如伪静态页面等，也可以使用分布式缓存来缓存较多的数据。应用通过高速局域网来访问，架构图如下图所示。



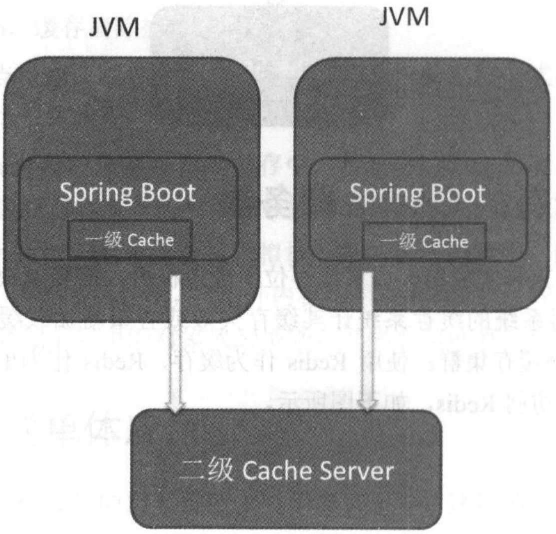

### 14.1.3 使用专有的 Cache 服务器

应用系统一般都是分布式应用，如果缓存位于虚拟机内，需要解决的问题是，当数据改变的时候，通知其他应用系统的缓存系统让其缓存失效或者重新加载缓存，Ehcache 就是通过 Terracotta 来组建的一个缓存集群。使用 Redis 作为缓存，Redis 作为单独的缓存服务器，分布式应用系统通过局域网访问 Redis，如下图所示。




## 14.1.4 使用一二级缓存服务器

使用 Redis 缓存，通过网络访问还是不如从内存中获取的性能好，所以通常称为二级缓存，从内存中取得缓存数据称为一级缓存。当应用系统需要查询缓存的时候，先从一级缓存里查找，如果有，则返回，如果没有查找到，则再查询二级缓存，架构图如下图所示。



## 14.2 Spring Boot Cache

Spring Boot 本身提供了一个基于 ConcurrentHashMap 的缓存机制，也集成了 EhCache2.x、JCache（JSR-107、EhCache3.x、Hazelcast、Infinispan），还有 Couchbase、Redis 等。Spring Boot 应用通过注解的方式使用统一的缓存，只需在方法上使用缓存注解即可，其缓存的具体实现依赖于选择的目标缓存管理器。使用@Cacheable 如下：

```
@Service
public class MenuServiceImpl implements MenuService {

    @Cacheable("menu")
    public Menu getMenu(Long id) {...}
}
```

MenuService 实例作为一个容器管理 Bean，Spring 将生成代理类，在实际调用 MenuService.getMenu 方法前，会调用缓存管理器，取得名为“menu”的缓存，此时，缓存项的 Key 就是方法参数 id，如果缓存命中，则返回此值，如果没有找到，则进入实际的 MenuService.getMenu 方法。在返回调用结果给调用者之前，还会将此查询结果缓存以备下次使用。

**集成 Spring Cache**

集成 Spring Cache，只需要在 pom 中使用以下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

如果使用 Spring 自带的内存的缓存管理器，需要在 appliaction.properties 中配置属性：

```
spring.cache.type=Simple
```

Simple 只适合单体应用或者开发环境使用，再或者是一个小微系统，通常你的应用是分布式应用，Spring Boot 还支持集成更多的缓存服务器。

- Simple，基于 ConcurrentHashMap 实现的缓存，适合单体应用或者开发环境使用。

- none，关闭缓存，比如开发阶段为了确保功能正确，可以先禁止使用缓存。
- redis，使用 Redis 作为缓存，还需要在 pom 中增加 Redis 依赖。本章将重点介绍 Redis 缓存以及扩展 Redis 实现一二级缓存。
- Generic，用户自定义缓存实现，用户需要实现一个 org.springframework.cache.CacheManager 的实现。
- 其他还有 JCache、EhCache 2.x、Hazelcast 等，不在这里一一介绍了。

> 为了简单起见，先使用 Simple 缓存。在 14.3 节将介绍 Redis 缓存。

最后，需要使用注解@EnableCaching 打开缓存功能。

```
@SpringBootApplication
@EnableCaching
public class Ch14Application {
  public static void main(String[] args) {
    SpringApplication.run(Ch14Application.class, args);
  }
}
```

## 14.3 注释驱动缓存

一旦配置好 Spring Boot 缓存，就可以在 Spring 管理的 Bean 中使用缓存注解，通常可以直接放在 Service 类上。

- @Cacheable，作用在方法上，触发缓存读取操作。
- @CacheEvict，作用在方法上，触发缓存失效操作。
- @CachePut，作用在方法上，触发缓存更新操作。
- @Cache，作用在方法上，综合上面的各种操作，在有些场景下，调用业务会触发多种缓存操作。
- @CacheConfig，在类上设置当前缓存的一些公共设置。

### 14.3.1 @Cacheable

注解 Cacheable 声明了方法的结果是可缓存的，如果缓存存在，则目标方法不会被调用，直接取出缓存。可以为方法声明多个缓存，如果至少有一个缓存有缓存项，则其缓存项将被返

回。代码如下：

```
@Cacheable({"Menu", "menuExt"})
public Menu findMenu(Long menuId) {...}
```

如果缓存不存在，则进入实际业务方法，将业务方法返回的结果缓存。

> **注意：** 对于不同的缓存实现，最好将缓存对象实现序列化 Serializable 接口，这样可以无缝切换到分布式缓存系统，比如：
>
> ```
> public class Menu implements Serializable{
>
> }
> ```

## 14.3.2 Key 生成器

缓存的 Key 非常重要，Spring 使用了 KeyGenerator 类来根据方法参数生成 Key，有以下规则。

如果只有一个参数，这个参数就是 Key。

```
@Cacheable(cacheNames="menu")
public Menu getMenu(Long id)
  ......
}
```

如果没有参数，则返回 SimpleKey.EMPTY，比如构造系统的菜单树，因为系统只有唯一的一个菜单树，所以不用指定参数，Key 值是 SimpleKey.EMPTY。

```
@Cacheable("menuTree")
public MenuNode getMenuTree() {
  .......
}
```

如果有多个 Key，则返回包含多个参数的 SimpleKey。

```
@Cacheable("user_function")
public boolean canAcessFunction(Long userId,Long orgId,String functionCode){
}
```

以上 canAcessFunction 用于判断登录用户 userId 所在的组织机构 orgId 是否具有访问 functionCode 的权限，此时 SimpleKey 包含了这三个字段。

Spring 使用 SimpleKeyGenerator 来实现上述 Key 的生成：

```
public class SimpleKeyGenerator implements KeyGenerator {
  @Override
  public Object generate(Object target, Method method, Object... params) {
    return generateKey(params);
  }

  public static Object generateKey(Object... params) {
    if (params.length == 0) {
      return SimpleKey.EMPTY;
    }
    if (params.length == 1) {
      Object param = params[0];
      if (param != null && !param.getClass().isArray()) {
        return param;
      }
    }
    return new SimpleKey(params);
  }
}
```

SimpleKey 实现了 hashCode 和 equals 方法：

```
this.hashCode = Arrays.deepHashCode(this.params);
@Override
public boolean equals(Object obj) {
  return (this == obj || (obj instanceof SimpleKey
        && Arrays.deepEquals(this.params, ((SimpleKey) obj).params)));
}
```

也可以实现自己的 KeyGenerator 方法，比如：

```
@Cacheable(cacheNames="org", keyGenerator="myKeyGenerator")
public Org findOrg(User  user) {...}
```

myKeyGenerator 实现了 KeyGenerator 接口，然后从 user 中获取到 orgId 作为缓存的 Key：

```
@Override
public Object generate(Object target, Method method, Object... params) {
  User user = (User)params[0];
  return user.getOrgId();
}
```

通常情况下，直接使用 SpEL 表达式来指定 Key 比自定义一个 KeyGenerator 更为简单：

```
@Cacheable(cacheNames="org", key="#orgId")
public Org findOrg(Long orgId, boolean checkCrmSystem) {...}

@Cacheable(cacheNames="org", key="#user.orgId")
public Org findOrg(User user) {...}
```

第一个方法仅仅使用参数 orgId 作为 Key，忽略参数 checkCrmSystem，第二个方法则获取 user 的 orgId 属性作为 Key。

也可以通过条件表达式来指定是否需要缓存，比如 orgId 大于 1000 的都需要缓存：

```
@Cacheable(cacheNames="org", condition="#orgId>10000")
public Org findOrg(Long orgId) {...}
```

还可以根据方法返回的结果来决定是否缓存，以下返回值如果 status 为 0，则不缓存：

```
@Cacheable(cacheNames="org", unless="#result.status==0")
public Org findOrg(Long orgId) {...}
```

### 14.3.3 @CachePut

注解 CachePut 总是会执行方法体，并且使用返回的结果更新缓存，同 Cacheable 一样，支持 condition、unless、key 选项，也支持 KeyGenerator。以下这个例子是更新组织机构：

```
@CachePut(cacheNames="org", key="#data.id")
public Org updateOrg(Org data){

}
```

## 14.3.4 @CacheEvict

注解 CacheEvict 用于删除缓存项或者清空缓存，CacheEvict 可以指定多个缓存名字来清空多个缓存，以下是修改用户基本信息缓存：

```
@CacheEvict(cacheNames="user",key="#id")
public void updateUser(Long id,int status){


}
```

清空“user”缓存中键值为 id 的缓存项。

> **注意**：CacheEvict 只具备删除缓存的功能，不具备加载缓存的功能，只有相应的@Cacheable 方法被调用后，才会加载最新缓存项。

CacheEvict 可以清空缓存中的所有项目，此时使用 allEntries=true 来删除清空缓存。重新加载配置后，会清空 config 缓存，代码如下：

```
@CacheEvict(cacheNames="config",allEntries=true)
public void loadConfig(){

}
```

以本章的菜单为例，如果添加菜单项目，通常会更新菜单相关的所有缓存：

```
@CacheEvict (cacheNames ={ "menu","menuTree"},allEntries=true)
public void addMenu(Menu menu) {

}
```

在更新菜单后，菜单树和菜单缓存都需要清空以便下次获取的时候重新构造。

### 14.3.5 @Caching

注解Caching可以混合以上各种注解，可以在Caching标签中混合@Cacheable、@CachePut、@CacheEvict。比如一个修改同时需要失效对应的用户缓存和用户扩展信息缓存。

```
@Caching(evict = { @CacheEvict(cacheNames="user", key="#user.id"),
@CacheEvict(cacheNames="userExt", key="#ext.id") })
public void updateUser(User user,UserExt ext){

}
```

### 14.3.6 @CacheConfig

到目前为止，所有的 Cache 注解都需要提供 Cache 名称，如果每个 Service 方法上都包含 Cache 名称，可能写起来重复。注解 CacheConfig 作用于类上，可以为此类的方法的缓存注解提供默认值，包括：

- 缓存的默认名称；
- 缓存的 KeyGenerator；

```
@CacheConfig("menu")
public class MenuServiceImpl implements MenuService
  @Cacheable()
  public Menu getMenu(Long id) {}
}
```

这样就不必为MenuService的每个缓存标签写一个cacheNames。

## 14.4 使用 Redis Cache

前面的例子都采用了缓存类型为Simple的Spring Boot自带的缓存实现。这个缓存与Spring Boot应用在同一个Java虚拟机内，适合单体应用系统。对于分布式应用，通常都会将缓存放在一台或者多台专门的缓存服务器上。本节开始介绍Redis Cache，它可用于分布式系统。

### 14.4.1 集成 Redis 缓存

在分布式应用中，使用Redis作为缓存是一种常用的选择(参看第12章了解Redis的使用)。

为了使用 Redis 缓存，需要在 pom 中引入 spring-boot-starter-data-redis：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

配置文件 application.properties 如下：

```
spring.cache.type=Redis
spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.password=Redis!123
```

第一行配置了 Redis 作为缓存，第二、第三行配置了 Redis 访问地址和端口，最后一行配置了 Redis 的访问密码。

配置完毕后，即可使用 Redis 缓存。

### 14.4.2 禁止缓存

通常在开发环境下，不需要使用 Redis 类型的缓存，你可以配置为 Simple，Spring Boot 自动切换，用 HashMap 作为缓存，或者使用 None 禁止缓存以方便开发。

```
#使用 Spring Boot 自带的缓存
#spring.cache.type=Simple
#禁止使用缓存采用 None
spring.cache.type=None
```

### 14.4.3 定制缓存

对于 RedisCacheManager 来说，还可以定制缓存项的存活时间，缓存名是否在 Redis 中加上前缀等，这是通过实现 CacheManagerCustomizer 配置来完成定制的：

```
@Configuration
public class RedisCacheManagerCustomizer {
  @Bean
```

```
  public CacheManagerCustomizer<RedisCacheManager> cacheManagerCustomizer() {
    return new CacheManagerCustomizer<RedisCacheManager>() {
      @Override
      public void customize(RedisCacheManager cacheManager) {
        Map<String, Long> expires = new HashMap<String, Long>();
        // 设置menu缓存，一分钟过期
        expires.put("menu", 60l);
        cacheManager.setExpires(expires);
      }

    };
  }

}
```

CacheManagerCustomizer 类提供了 customize 回调方法，回调代码为缓存 menu 定义了一个 60 秒超时。

## 14.5 Redis 缓存原理

> 对于使用 Spring Boot 缓存，熟悉本章前 3 节知识就可以了，如果还想深入 Spring Cache 原理，以及实现性能更好一级二级缓存，可以从这一节开始深入了解。阅读前，最好已经掌握了 Spring Boot 自动装配技术，可以参考第 7 章了解 Spring Boot 自动装配。

Spring Boot 的自动配置类 RedisCacheConfiguration 会自动检测 spring.cache.type 的值，从而决定使用何种缓存。

```
@Configuration
@AutoConfigureAfter(RedisAutoConfiguration.class)
@ConditionalOnBean(RedisTemplate.class)
@ConditionalOnMissingBean(CacheManager.class)
@Conditional(CacheCondition.class)
class RedisCacheConfiguration {

}
```

下面逐一解释 RedisCacheConfiguration 的注解。

```
@AutoConfigureAfter(RedisAutoConfiguration.class)
```

```
@ConditionalOnBean(RedisTemplate.class)
```

基于 Redis 的缓存首先必须要确保 Redis 配置成功，Redis 缓存会使用 RedisTemplate 来操作缓存，因此 RedisCacheConfiguration 类使用@AutoConfigureAfter 和@ConditionalOnBean 注解。

其次，ConditionalOnMissingBean 注解意味着该配置生效的条件是系统还没有 CacheManager 的实现。

```
@ConditionalOnMissingBean(CacheManager.class)
```

最后，注解 Conditional 需要联合同一个 CacheCondition 来进一步判断是否使用执行当前配置类：

```
@Conditional(CacheCondition.class)
```

CacheCondition 实现了 Condition 接口，该接口返回一个 boolean 值，用于判断是否能启用该配置。CacheCondition 用于判断 application.properties 中是否配置了 spring.cache.type，取出其对应的值与 CacheCondition 所在的配置类进行比较，如果一致，则返回 match。

RedisCacheConfiguration 提供了配置 RedisCacheManager 的实现：

```
@Bean
public RedisCacheManager cacheManager(RedisTemplate<Object, Object>
redisTemplate) {
    RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
    .....
    return cacheManager;
}
```

RedisCacheManager 用于实现 Redis 的缓存管理，RedisCacheManager 实现了 CacheManager 接口，CacheManager 有如下定义：

```
public interface CacheManager {
    Cache getCache(String name);
    Collection<String> getCacheNames();
}
```

getCache 方法会根据缓存名字得到缓存，getCacheNames 则返回 CacheManager 知道的缓存。RedisCacheManager 将返回 RedisCache，其实现了 Cache 接口，类似 Java.util.Map 接口，包含以

下方法：

- ValueWrapper get(Object key)，根据 Key 值获取缓存对象，ValueWrapper 包含了 get 方法用于获取缓存对象。
- void put(Object key, Object value)，设置缓存。
- void evict(Object key)，根据 Key 值清除缓存。
- void clear()，清除所有缓存。

RedisCache 使用 RedisTemplate 来存放和读取缓存，将在下一节进一步讲述。

## 14.6 实现 Redis 两级缓存

Spring Boot 自带的 Redis 缓存非常容易使用，但由于通过网络访问了 Redis，效率还是比传统的跟应用部署在一起的一级缓存略慢。本节将扩展 RedisCacheManager 和 RedisCache，在访问 Redis 之前，先访问一个 ConcurrentHashMap 实现的简单一级缓存，如果有缓存项，则返回给应用，如果没有，再从 Redis 中取得，并将缓存对象放到一级缓存中。

当缓存项发生变化的时候，注解@CachePut 和@CacheEvict 会触发 RedisCache 的 put( Object Key, Object Value)和 evict(Object Key)操作，两级缓存需要同时更新 ConcurrentHashMap 和 Redis 缓存，而且需要通过 Redis 的 Pub 发出通知消息，其他 Spring Boot 应用通过 Sub 来接收消息，同步更新 Spring Boot 应用自身的一级缓存。

### 14.6.1 实现 TwoLevelCacheManager

首先，创建一个新的缓存管理器，命名为 TwoLevelCacheManager，继承了 Spring Boot 的 RedisCacheManager，重载 decorateCache 方法。返回的是新创建的 LocalAndRedisCache 缓存实现。

```
class TwoLevelCacheManager extends RedisCacheManager {
    RedisTemplate redisTemplate;
    public TwoLevelCacheManager(RedisTemplate redisTemplate,RedisCacheWriter
cacheWriter, RedisCacheConfiguration defaultCacheConfiguration) {
        super(cacheWriter,defaultCacheConfiguration);
        this.redisTemplate = redisTemplate;
    }
    //使用 RedisAndLocalCache 代替 Spring Boot 自带的 RedisCache
```

```
    @Override
    protected Cache decorateCache(Cache cache) {
      return new RedisAndLocalCache(this, (RedisCache) cache);
    }

  public void publishMessage(String cacheName) {
    this.redisTemplate.convertAndSend(topicName, cacheName);
  }
  // 接受一个消息清空本地缓存
  public void receiver(String name) {
    RedisAndLocalCache cache = ((RedisAndLocalCache) this.getCache(name));
    if(cache!=null){
      cache.clearLocal();
    }
  }

}
```

在 Spring Cache 中，在缓存管理器中创建好每个缓存后，都会调用 decorateCache 方法，这样缓存管理器子类就有机会实现自己的扩展。在这段代码中，返回了自定义的 RedisAndLocalCache 实现。publishMessage 方法提供给 Cache，用于在缓存更新时使用 Redis 的消息机制通知其他分布式节点的一级缓存。receiver 方法对应于 publishMessage 方法，当收到消息后，会清空一级缓存。

## 14.6.2 创建 RedisAndLocalCache

RedisAndLocalCache 是系统的核心，它实现了 Cache 接口、类，会实现如下操作。

- get 操作，通过 Key 取出对应的缓存项，在调用父类 RedisCache 之前，会先检测本地缓存是否存在，存在则不需要调用父类的 get 操作；如果不存在，则调用父类的 get 操作后，将 Redis 返回的 ValueWrapper 放到本地缓存中待下次使用。
- put，调用父类 put 操作更新 Redis 缓存，同时广播消息，缓存改变。我们将在后面讲解如何使用 Redis 的 Pub/Subscribe 来同步缓存。
- evict，同 put 操作一样，调用父类处理，清空对应的缓存，同时广播消息。
- putIfAbsent，同 put 操作一样，调用父类实现，同时广播消息。

RedisAndLocalCache 的构造如下：

```
class RedisAndLocalCache implements Cache {
  // 本地缓存提供
  ConcurrentHashMap<Object, Object> local = new ConcurrentHashMap<Object,
Object>();
  RedisCache redisCache;
  TwoLevelCacheManager cacheManager;

  public RedisAndLocalCache(TwoLevelCacheManager cacheManager, RedisCache
redisCache) {
    this.redisCache = redisCache;
    this.cacheManager = cacheManager;
  }

  @Override
  public String getName() {
    return redisCache.getName();
  }

  @Override
  public Object getNativeCache() {
    return redisCache.getNativeCache();
  }

  // 其他 get、put、evict 方法参考后面代码片段说明
}
```

如上述代码所示，RedisAndLocalCache 实现了 Cache 接口，并使用了真正的 RedisCache 作为其实现方法，其关键的 get 和 put 方法如下：

```
@Override
public ValueWrapper get(Object key) {
  // 一级缓存先取
  ValueWrapper wrapper = (ValueWrapper) local.get(key);
  if (wrapper != null) {
    return wrapper;
  } else {
    // 二级缓存先取
```

```
      wrapper = redisCache.get(key);
      if (wrapper != null) {
        local.put(key, wrapper);
      }
      return wrapper;
    }
  }

  @Override
  public void put(Object key, Object value) {
    System.out.println(value.getClass().getClassLoader());
    redisCache.put(key, value);
    // 通知其他节点缓存更新
    clearOtherJVM();
  }
  @Override
  public void evict(Object key) {
    redisCache.evict(key);
    // 通知其他节点缓存更新
    clearOtherJVM();
  }
  protected void clearOtherJVM() {
     cacheManager.publishMessage(redisCache.getName());
  }
  // 提供给 CacheManager 清空一级缓存
  public void clearLocal() {
    this.local.clear();
  }
```

变量 local 代表了一个简单的缓存实现，使用了 ConcurrentHashMap，其 get 方法有如下逻辑实现：

- 通过 Key 从本地取出 ValueWrapper；
- 如果 ValueWrapper 存在，则直接返回；
- 如果 ValueWrapper 不存在，则调用父类 RedisCache 取得缓存项；
- 如果缓存项为空，则说明暂时无此项，直接返回空，等待@Cacheable 调用业务方法获取缓存项。

put 方法的实现逻辑如下：

- 先调用 redisCache，更新二级缓存；
- 调用 clearOtherJVM 方法，通知其他节点缓存更新；
- 其他节点（包括本节点）的 TwoLevelCacheManager 收到消息后，会调用 receiver 方法从而实现一级缓存。

为了简单起见，一级缓存的同步更新仅仅是清空一级缓存，并非采用同步更新缓存项。一级缓存将在下一次 get 方法调用时再次从 Redis 中加载最新数据。

一节缓存仅仅简单使用了 Map 实现，并未实现缓存的多种策略。因此，如果你的一级缓存需要各种缓存策略，还需要用一些第三方库或者自行实现，但大部分情况下 TwoLevelCacheManager 都足够使用。

## 14.6.3 缓存同步说明

当缓存发生改变的时候，需要通知分布式系统的 TwoLevelCacheManager 清空一级缓存。这里使用 Redis 实现消息通知，关于 Redis 消息发布和订阅，请参考 Redis 一章。

为了实现 Redis 的 Pub/Sub 模式，我们需要在 CacheConfig 中添加一些代码，创建一个消息监听器：

```
 // 定义一个 Redis 的频道，默认叫 cache，用于 pub/sub
 @Value("${springext.cache.redis.topic:cache}")
 String topicName;
 // Redis message，参考 Redis 一章
 @Bean
RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory,
                                      MessageListenerAdapter listenerAdapter) {
  RedisMessageListenerContainer container = new RedisMessageListenerContainer();
  container.setConnectionFactory(connectionFactory);
  container.addMessageListener(listenerAdapter, new PatternTopic(topicName));
  return container;
 }
```

如上所示，需要在配置文件中配置 springext.cache.redis.topic，指定一个频道的名字，如果

没有配置，默认的频道名称是 cache。

配置一个监听器很简单，只需要实现 MessageListenerAdapter，并注册到 RedisMessageListenerContainer 即可。

MessageListenerAdapter 需要实现 onMessage 方法，我们只需要获取消息内容，这里是指要清空的缓存名字，然后交给 MyRedisCacheManager 处理即可

```
@Bean
MessageListenerAdapter listenerAdapter(final TwoLevelCacheManager cacheManager) {
  return new MessageListenerAdapter(new MessageListener() {
    public void onMessage(Message message, byte[] pattern) {
      byte[] bs = message.getChannel();
      try {
        // Sub 一个消息，通知缓存管理器，这里的 type 就是 Cache 的名字
        String type = new String(bs, "UTF-8");
        cacheManager.receiver(type);
      } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
        // 不可能出错，忽略
      }
    }
  });
}
```

## 14.6.4 将代码组合在一起

前面分别实现了缓存管理器、缓存，以及缓存之间的同步，现在需要将缓存管理器配置为应用的缓存管理器，通过搭配@Configuration 和@Bean 实现：

```
@Configuration
public class CacheConfig {
  @Bean
  public TwoLevelCacheManager cacheManager(RedisTemplate redisTemplate) {
    // RedisCache 需要一个 RedisCacheWriter 来实现读写 Redis
    RedisCacheWriter writer = RedisCacheWriter.lockingRedisCacheWriter
(redisTemplate.getConnectionFactory());
    /*SerializationPair 用于 Java 和 Redis 之间的序列化和反序列化，我们这里使用自带
```

```
的 JdkSerializationRedisSerializer，并在反序列化过程中，使用当前的 ClassLoader*/
        SerializationPair pair = SerializationPair.fromSerializer(new
JdkSerializationRedisSerializer(this.getClass().getClassLoader()));
        /*构造一个 RedisCache 的配置，比如是否使用前缀，比如 Key 和 Value 的序列化机制*/
        RedisCacheConfiguration config = RedisCacheConfiguration.
defaultCacheConfig().serializeValuesWith(pair);
        /*创建 CacheManager，并返回给 Spring 容器*/
        TwoLevelCacheManager cacheManager = new TwoLevelCacheManager
(redisTemplate,writer,config);
        return cacheManager;
    }
}
```

构造一个 TwoLevelCacheManager 较为复杂，这是因为构造 RedisCacheManager 复杂导致的，构造 RedisCacheManager 需要如下两个参数：

- RedisCacheWriter，一个实现 Redis 操作的接口，Spring Boot 提供了 NoLock 和 Lock 两种实现，在缓存写操作的时候，前者有较高性能，而后者实现了 Redis 锁。
- RedisCacheConfiguration，用于设置缓存特性，比如缓存项目的 TTL（存活时间）、缓存 Key 的前缀等，默认情况下 TTL 为 0，不使用前缀。你可以为缓存管理器设置默认的配置，也可以为每一个缓存设置一个配置。最为重要的配置是 SerializationPair，用于 Java 和 Redis 的序列化和反序列化操作，这里使用自带的 JdkSerializationRedisSerializer 作为序列化机制，这个类在 Reids 一章有详细介绍。

以上代码实现了一二级缓存，行数不到 200 行代码。相对于自带的 RedisCache 来说，缓存效率更高。相对于专业的一二级缓存服务器来说，如 Ehcache+Terracotta 组合，更加轻量级。

# 第 15 章 Spring Session

Spring Boot 应用通常会部署在多个 Web 服务器上同时提供服务，这样做有很多好处：

- 单个应用宕机不会停止服务，升级应用可以逐个升级而不必停止服务。
- 提高了应用整体的吞吐量。

我们称这种部署方式为水平扩展，前端通过 Nginx 提供反向代理，会话管理可以通过 Spring Session，使用 Redis 来存放 Session。部署 Spring Boot 应用到任意一台 Web 服务器上，从而提高了系统可靠性和可伸缩性。

## 15.1 水平扩展实现

当系统想提升处理能力的时候，通常用两种选择，一种是重置扩展架构，即提升现有系统硬件的处理能力，比如提高 CPU 频率、使用更好的存储器。另外一种选择是水平扩展架构，即部署系统到更多的服务器上同时提供服务。这两种方式各有利弊，现在通常都优先采用水平扩展架构，这是因为：

- 重置扩展架构

  缺点：架构中的硬件提升能力有限，而且硬件能力提升往往需要更多的花销；
  优点：应用系统不需要做任何改变。

- 水平扩展

  优点：成本便宜；
  缺点：更多的应用导致管理更加复杂。对于 Spring Boot 应用，会话管理是一个难点。

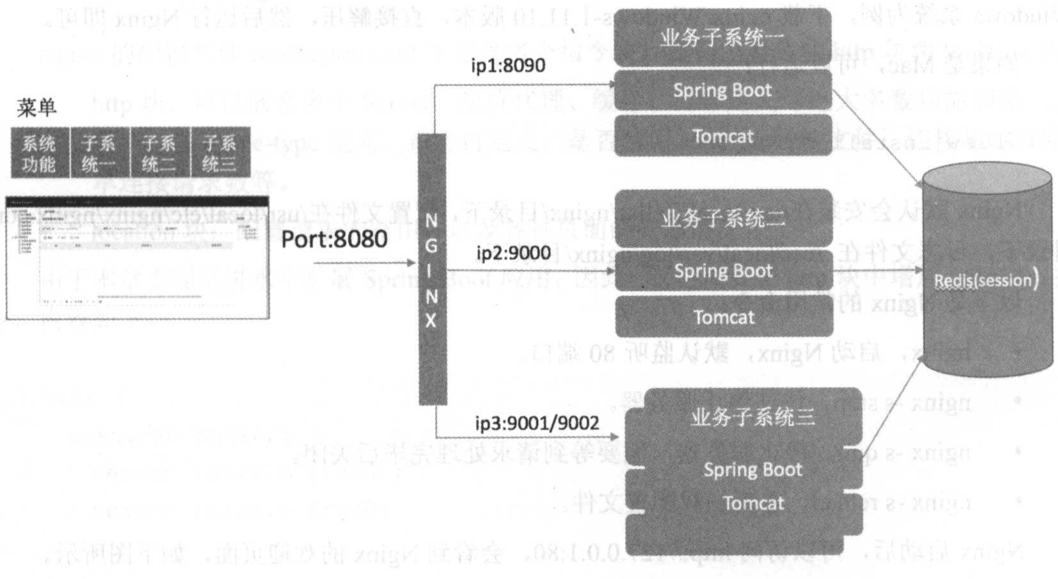
Spring Boot 应用水平扩展有两个问题需要解决，一个是将用户的请求派发到水平部署的任意一台 Spring Boot 应用，通常用一个反向代理服务器来实现，本书将使用 Nginx 作为反向代理服务器。

> 反向代理（Reverse Proxy）方式是指接收 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。
>
> 正向代理服务器：局域网内通过一个正向代理服务器访问外网。

另外一个需要解决的问题是会话管理，单个 Spring Boot 应用的会话由 Tomcat 来管理，会话信息与 Tomcat 存放在一起。如果部署多个 Spring Boot 应用，对于同一个用户请求，即使请求通过 Nginx 派发到不同的 Web 服务器上，也能共享会话信息。有两种方式可以实现。

- 复制会话：Web 服务器通常都支持 Session 复制，一台应用的会话信息改变将立刻复制到其他集群的 Web 服务器上。
- 集中式会话：所有 Web 服务器都共享一个会话，会话信息通常存放在一台服务器上，本章使用 Redis 服务器来存放会话。

复制会话的缺点是每次会话改变需要复制到多台 Web 服务器上，效率较低。因此 Spring Boot 应用采用第二种方式（集中式会话方式），结构如下图所示。




上图是一个大型分布式系统架构，包含了三个独立的子系统。业务子系统一和业务子系统

二分别部署在一台 Tomcat 服务器上，业务子系统三部署在两台 Tomcat 服务器上，采用水平扩展。

架构采用 Nginx 作为反向代理，其后的各个子系统都采用 Spring Session，将会话存放在 Redis 中，因此，这些子系统虽然是分开部署的，支持水平扩展，但能整合成一个大的系统。Nginx 提供统一的入口，对于用户访问，将按照某种策略，比如根据访问路径派发到后面对应的 Spring Boot 应用中，Spring Boot 调用 Spring Session 取得会话信息，Spring Session 并没有从本地存取会话，会话信息存放在 Redis 服务器上。

## 15.2 Nginx 的安装和配置

Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）、TCP/UDP 代理服务器，并在一个 BSD-like 协议下发行。由俄罗斯的程序设计师 Igor Sysoev 开发，供俄国大型的入口网站及搜索引擎 Rambler 使用。其特点是占有内存少，并发能力强，事实上 Nginx 的并发能力确实在同类型的网页服务器中表现较好，国内使用 Nginx 的网站有百度、新浪、网易、腾讯等。

### 15.2.1 安装 Nginx

打开 Nginx 网站（http://nginx.org/），进入下载页面，根据自己的操作系统选择下载，以 Windows 系统为例，下载 nginx/Windows-1.11.10 版本，直接解压，然后运行 Nginx 即可。

如果是 Mac，可以运行：

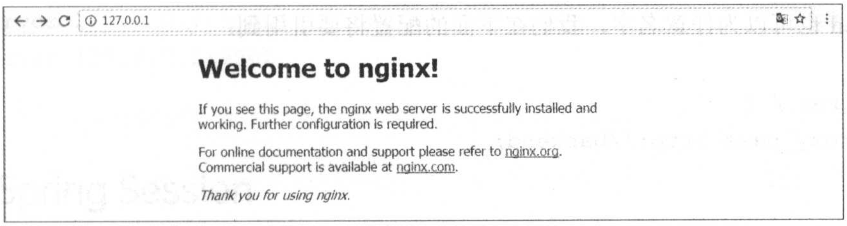
```
>brew install nginx
```

Nginx 默认会安装在/usr/local/Cellar/nginx/目录下，配置文件在/usr/local/etc/nginx/nginx.conf 目录下，日志文件在 /usr/local/var/log/nginx/目录下。

以下是 Nginx 的常用命令：

- nginx，启动 Nginx，默认监听 80 端口。
- nginx -s stop，快速停止服务器。
- nginx -s quit，停止服务器，但要等到请求处理完毕后关闭。
- nginx -s reload，重新加载配置文件。

Nginx 启动后，可以访问 http://127.0.0.1:80，会看到 Nginx 的欢迎页面，如下图所示。




如果 80 端口访问不了，则可能是因为你下载的版本的原因，Nginx 的 HTTP 端口配置成其他端口，编辑 conf/nginx.conf，找到：

```
server {
  listen       80;
}
```

修改 listen 参数到 80 端口即可。

Nginx 的 log 目录下提供了三个文件：

- access.log，记录了用户的请求信息和响应。
- error.log，记录了 Nginx 运行的错误日志。
- nginx.pid，包含了 Nginx 的进程号。

## 15.2.2 配置 Nginx

Nginx 的配置文件 conf/nginx.conf 下包含多个指令块，我们主要关注 http 块和 location 块。

- http 块：可以嵌套多个 Server，配置代理、缓存、日志定义等绝大多数功能和第三方模块，如 mime-type 定义、日志自定义、是否使用 sendfile 传输文件、连接超时时间、单连接请求数等。
- location 块：配置请求的路由，以及各种页面的处理情况。

由于本章主要是讲水平扩展 Spring Boot 应用，因此，我们需要在 http 块中增加 upstream 指令，内容如下：

```
http {
  upstream backend {
    server 127.0.0.1:9000;
    server 127.0.0.1:9001
  }
}
```

backend 也可以为任意名字，我们在下面的配置将要引用到：

```
location / {
   proxy_pass http://backend;
}
```

location 后可以是一个正则表达式，我们这里用 “/” 表示所有客户端请求都会传给 http://backend，也就是我们配置的 backend 指令的地址列表。因此，整个 http 块类似下面的样子：

```
http {
   include       mime.types;
   default_type  application/octet-stream;
   sendfile       on;
   keepalive_timeout  65;
   upstream backend {
     server 127.0.0.1:9000;
     server 127.0.0.1:9001;
   }
   server {
      listen       80;
      server_name  localhost;

      location / {
           proxy_pass http://backend;
      }
   }

}
```

我们在后面将创建一个 Spring Boot 应用，并分别以 9000 和 9001 两个端口启动，然后在 Spring Session 的基础上一步步来完成 Spring Boot 应用的水平扩展。

> **注意：** Nginx 反向代理默认情况下会轮询后台应用，还有一种配置是设置 ip_hash，这样，固定客户端总是反向代理到后台的某一个服务器。这种设置方式就不需要使用 Spring Session 来管理会话，使用 Tomcat 的会话管理即可。但弊端是如果服务器宕机或者因为维护重启，则会话丢失。ip_hash 设置如下：
>
> ```
> upstream backend {
>  ip_hash;
> ```

```
    server 127.0.0.1:9000;
    server 127.0.0.1:9001
}
```

## 15.3 Spring Session

### 15.3.1 Spring Session 介绍

在默认情况下，Spring Boot 使用 Tomcat 服务器的 Session 实现，我们编写一个例子用于测试：

```
@Controller
public class SpringSessionCrontroller {

    Log log = LogFactory.getLog(SpringSessionCrontroller.class);

    @RequestMapping("/putsession.html")
    public @ResponseBody String putSession(HttpServletRequest request){
        HttpSession session = request.getSession();
        log.info(session.getClass());
        log.info(session.getId());
        String name = "xiandafu";
        session.setAttribute("user", name);
        return "hey,"+name;
    }
}
```

如果访问服务/putsession.html，控制台输出为：

```
SpringSessionCrontroller          : class
org.apache.catalina.session.StandardSessionFacade
SpringSessionCrontroller          : F567C587EA25CBD5B9A75C62AB51904D
```

可以看到，Session 管理是通过 Tomcat 提供的 org.apache.catalina.session.StandardSessionFacade 实现的。

在配置文件 application.properties 中添加如下内容：

```
spring.session.store-type=Redis|JDBC|Hazelcast|none
```

Spring Boot 配置很容易切换到不同的 Session 管理方式，总共有以下几种：

- Redis，Session 数据存放 Redis 中，本节将重点介绍。
- JDBC，会话数据存放在数据库中，默认情况下 SPRING_SESSION 表存放 Session 基本信息，如 sessionId、创建时间、最后一次访问时间等，SPRING_SESSION_ATTRIBUTES 存放了 session 数据，ATTRIBUTE_NAME 列保存了 Session 的 Key，ATTRIBUTE_BYTES 列以字节形式保存了 Session 的 Value，Spring Session 会自动创建这两张表。
- Hazelcast，Session 数据存放到 Hazelcast。
- None，禁用 Spring Session 功能。

通过配置属性 spring.session.store-type 来指定 Session 的存储方式，如：

```
spring.session.store-type=Redis
```

修改为配置和增加 Spring Session 依赖后，如果访问服务/putsession.html，控制台输出为：

```
SpringSessionCrontroller          : class
org.springframework.session.web.http.SessionRepositoryFilter$SessionReposito
ryRequestWrapper$HttpSessionWrapper

SpringSessionCrontroller          :d4315e92-48e1-4a77-9819-f15df9361e68
```

可以看到，Session 已经替换为 HttpSessionWrapper 实现，这个类负责 Spring Boot 的 Session 存储类型的具体实现。

### 15.3.2 使用 Redis

本章将用 Redis 来保存 Session，你需要安装 Redis，如果未安装，请参考 Redis 一章，Spring Boot 的配置如下：

```
spring.session.store-type=Redis
spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.password=Redis!123
```

还需要引入对 Redis 的依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

再次访问/putsession.html 后，我们通过 Redis 客户端工具访问 Redis，比如使用 redis-cli，输入如下命令：

```
keys spring:session:*
```

查询所有“spring:session:”开头的 keys，输出如下：

```
3) "spring:session:sessions:expires:863c7e73-8249-4780-a08e-0ff2bddda86"
...
7) "spring:session:sessions:863c7e73-8249-4780-a08e-0ff2bddda86"
```

会话信息存放在“spring:session:sessions:”开头的 Key 中，863c7e73-8249-4780-a08e-0ff2bddda86 代表一个会话 id，“spring:session:sessions”是一个 Hash 数据结构，可以用 Redis HASH 相关的命令来查看这个用户会话的数据，使用 hgetall 查看会话所有的信息：

```
>hgetall "spring:session:sessions:863c7e73-8249-4780-a08e-0ff2bddda86"
1) "sessionAttr:user"
2) "maxInactiveInterval"
.......
```

使用以下命令来查看该 Session 的 user 信息：

```
>HMGET "spring:session:sessions:863c7e73-8249-4780-a08e-0ff2bddda86"
sessionAttr:user
```

sessionAttr:user 是 Spring Session 存入 Redis 的 Key 值，sessionAttr:是其前缀，user 是我们在 Spring Boot 中设置会话的 Key。其他 Spring Boot 默认创建的 Key 还有：

- creationTime，创建时间。
- maxInactiveInterval，指定过期时间（秒）。

- lastAccessedTime，上次访问时间。
- sessionAttr，以“sessionAttr:”为前缀的会话信息，比如 sessionAttr: user。

因此，Spring Session 使用 Redis 保存的会话将采用如下的 Redis 操作，类似如下：

```
>HMSET spring:session:sessions:863c7e73-8249-4780-a08e-0ff2bddda86
creationTime 1404360000000 maxInactiveInterval 1800 lastAccessedTime
1404360000000 sessionAttr:attrName someAttrValue sessionAttr:attrName2
someAttrValue2
```

> **注意：**Spring Session 的 Redis 实现并不是每次通过 Session 类获取会话信息或者保存的时候都会调用 Redis 操作，它会先尝试从内部的 HashMap 读取值，如果没有，才调用 Redis 的 HMGET 操作。同样，当保存会话的时候，也没有立即调用 Redis 操作，而是先保存到 HashMap 中，等待服务请求结束后再将变化的值使用 HMSET 更新。如果你想在保存会话操作后立即更新到 Redis 中，需要配置成 IMMEDIATE 模式，修改配置属性：
>
> ```
> spring.session.redis.flushMode=IMMEDIATE
> ```

我们注意到，还有另外一个 Redis Key 是“spring:session:sessions:expires:863c7e73-8249-4780-a08e-0ff2bddda86”，这是因为 Redis 会话过期并没有直接使用在 session:sessions:key 变量上，而是专门用在 session:sessions:expires:key 上，当此 Key 过期后，会自动清除对应的会话信息。使用 ttl 查看会话过期时间：

```
>ttl spring:session:sessions:expires:863c7e73-8249-4780-a08e-0ff2bddda86
(integer) 1469
```

默认是 1800 秒，即 30 分钟，现在只剩下 1469 秒。

### 15.3.3 Nginx+Redis

在 15.2.2 节中，我们已经配置了：

```
upstream backend {
  server 127.0.0.1:9000;
  server 127.0.0.1:9001
}
```

假设在本机上部署了两个 Spring Boot 应用，使用端口分别是 9000 和 9001。进入工程目录，运行 mvn package，我们看到 ch15.springsession\target\目录下生成了 ch17.springsession-0.0.1-SNAPSHOT.jar。然后进入命令行，进入 target 目录，启动这个 Spring Boot 应用：

```
java -jar target/ch15.springsession-0.0.1-SNAPSHOT.jar  --server.port=9000
```

打开另外一个命令窗口，进入工程目录，运行：

```
java -jar target/ch15.springsession-0.0.1-SNAPSHOT.jar  --server.port=9001
```

这时候，我们就有两台 Spring Boot 应用。接下来，我们访问以下地址，并刷新多次：

```
http://127.0.0.1/putsession.html
```

这时候就看到两个 Spring Boot 应用均有日志输出，比如 9000 端口的应用控制台输出如下：

```
class org.springframework.session.web.http.SessionRepositoryFilter....
863c7e73-8249-4780-a08e-0ff2bdddda86
```

9001 端口的 Spring Boot 应用也有类似输出：

```
class org.springframework.session.web.http.SessionRepositoryFilter...
863c7e73-8249-4780-a08e-0ff2bdddda86
```

我们看到，两个 Spring Boot 应用都具有相同的 sessionId，如果停掉任意一台应用，系统还有另外一台服务器提供服务，会话信息保存在 Redis 中。

# 第 16 章 Spring Boot 和 ZooKeeper

前面一章使用 Spring Session 实现了 Spring Boot 水平扩展，每个 Spring Boot 应用与其他水平扩展的 Spring Boot 一样，都能处理用户请求。如果宕机，Nginx 会将请求反向代理到其他运行的 Spring Boot 应用上，如果系统需要增加吞吐量，只需要再启动更多的 Spring Boot 应用即可。

在企业应用中，实现了 Spring Boot 水平扩展后，我们还面临一些挑战：

- 一个时刻，只能有一个应用来处理某个业务，而不能发生同时处理的情况。这就需要提供一个分布式锁，只有获得锁的 Spring Boot 应用才能执行操作。执行完毕后，释放锁。比如定时调度某个任务执行，同一个时刻只能有一个 Spring Boot 应用能执行。
- 应用系统之间通过 REST 接口来相互调用，如何让 REST 客户端知道服务在哪里？Spring Boot 应用可以水平扩展，不可能通过 application.properties 来配置服务器地址。
- 需要在 Spring Boot 应用中选择一个领导节点，这个领导节点负责协调所有 Spring Boot 应用的工作。

Spring Boot 并未提供如上所述的协调能力，有些开源产品已经提供了分布式协调能力，本章要介绍的 ZooKeeper 就是这样一款协调器。协调器本身也是分布式的，以保证协调器的高可用，所以也称为分布式协调器。分布式协调器是分布式系统和大数据系统必备的一个基础服务。

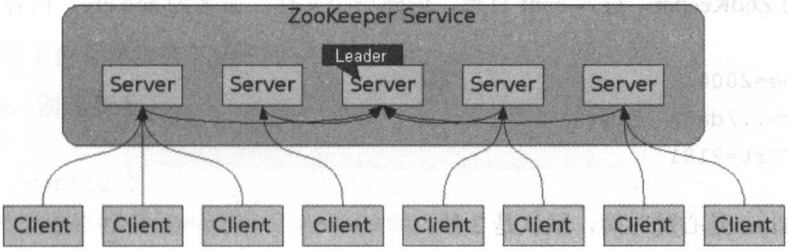
# 16.1 ZooKeeper

ZooKeeper 来源于 Apache Hadoop 子项目，是一个高性能、分布式的、开源应用协调服务。分布式应用可以基于它实现协调服务，比如同步、集群、领导选取，以及分布式系统的配置管理、命名服务。它被设计为易于编程，使用文件系统目录树作为数据模型。服务端跑在 Java 上，提供 Java 和 C 的客户端 API。

ZooKeeper（下面简称 zk）有以下特点：

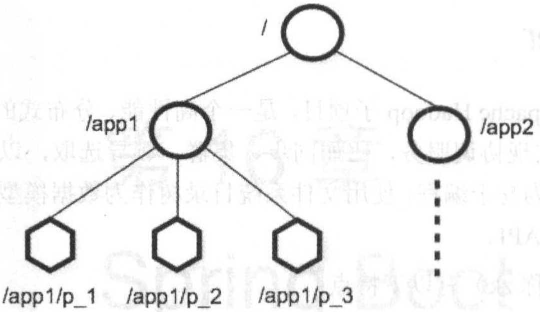
- 简单的 API 和数据结构完成协调服务，zk 提供了易于理解的数据结构来完成协调服务，其 Java API 非常简单。Curator 进一步封装了这些 API，直接提供了分布式协调服务而不需要关心细节。
- 分布式，不会出现单点故障，一般来说，至少部署 3 台 zk 以避免单点故障。客户端（指 Spring Boot 应用）如果连接的 zk 宕机，客户端将自动连接到另外一台。
- 保证操作的时序性，zk 对每次更新都有时间戳记录，从而保证操作的时序性，保证可以完成更高层次的协调服务，如分布式锁。
- 性能测试结果，zk 本身的性能非常好，既可以处理分布式系统的管理协调任务，如选举领导，也能胜任高并发量的业务协调处理，如业务处理的分布式锁。

下图中 Client 就是我们的 Spring Boot 应用，Server 表示 zk，多台 zk 协作保证了 zk 的高可用性，提供了协调服务。Spring Boot 应用连接一个 zk，如果这个 zk 变得不可用，则会连接另外一个。




## 16.1.1 ZooKeeper 的数据结构

zk 提供的命名空间（name space）类似文件系统，每一个节点都是通过路径来表示的，不同的是，节点可以包含一定的数据（2MB 字节），这些节点可以用来存放业务信息，如配置信息等，如下图所示。




节点还包含了更新的版本、时间戳。有一种特殊的节点是临时节点，创建节点的会话存在，节点就存在，一旦会话结束，如客户端创建的连接断掉，或者客户端主动关闭此会话，则节点会被删除。还可以指定节点为顺序节点，创建节点的时候，自动为节点增加一个序列号，并且序列号递增。

节点可以被监控，一旦节点变化，如删除节点，或者节点数据变化，客户端就会收到此事件，此监控失效。客户端可以调用 API 继续监控这个节点。

## 16.1.2 安装 ZooKeeper

进入 ZooKeeper 官网 https://zookeeper.apache.org/，从首页找到下载链接，进入下载页面，选择最新版本下载。建议下载前选择国内镜像，这样下载速度较快。

zk 只需要解压即可使用。

为了启动 ZooKeeper，进入 conf 目录，创建一个文件，命名为 zoo.cfg，内容如下：

```
tickTime=2000
dataDir=../data
clientPort=2181
```

- tickTime 是心跳时间，默认是 2 秒。

> 心跳是指像客户端按照一定频率发送心跳包到服务器端以维持网络套接字连接，心跳包告诉服务器，客户端还在运行。心跳包通常用于长连接，如果长连接没有心跳包，会导致服务器或者防火墙主动断开。心跳包通常就是内容为空的包。

- dataDir 是 ZooKeeper 保存的内存快照，以及事务日志的目录。
- clientPort 是客户端应用连接的端口。

然后进入 Bin 目录，运行 zkServer：

```
D:\apache\zookeeper-3.4.8\bin>zkServer
```

使用 Ctrl+C 停止 zkServer。

在 Linux 中，启动 zk 的命令是：

```
[root@sample bin]# ./zkServer.sh start
```

停止使用./zkServer.sh，使用 stop 命令即可。

> 默认情况下，zkEnv 会设置默认的配置文件为 zoo.cfg，如果你需要使用其他配置文件，可以修改 zkEnv.cmd 或者 zkEnv.sh。

### 16.1.3 ZooKeeper 的基本命令

服务器启动成功，我们可以运行 zkCli 来连接到 ZooKeeper 服务器上进行操作。

- help，进入命令行模式，直接键入 help，可以查看命令帮助，本书列举了一些常用命令。
- ls，查看目录。

```
> ls /
```

以上命令用于查看根目录下的节点。

- create，创建节点。

```
>create /config 0
>create /config/db.username xiandafu
```

创建一个 config 节点，节点内容是字符“0”，第二条在 config 目录下创建一个节点 db.username，节点的数据是字符“xiandafu”。

- create -e，创建临时节点，一旦用户会话结束，则节点自动删除。

```
> create /server data
> create -e /server/s1 192.168.0.2
```

上面创建了一个 server 目录，节点数据是任意数据（这里用 data 表示任意数据），然后在其目录下创建一个临时节点，用参数-e 表示，节点名字是 s1，数据是 IP 地址 192.168.0.2。

如果会话连接还在，用 zkCli 打开另外一个命令行窗口，在控制台 B 中查看结果：

```
>ls /server
[s1]
```

说明 server 下有 s1 节点，如果此时关闭了创建 s1 节点的会话窗口 A，然后再次在控制台 B 上运行此命令：

```
>ls /server
[]
```

发现 s1 节点已经被自动删除。

- create -s，创建带有序列号的节点，带有-s 的 create 命令会自动为节点增加一个序列号递增后缀，可以通过这个判断节点创建的先后顺序。

```
>create /task data
>create -s /task/t data
>create -s /task/t data2
```

先创建一个 task 节点，数据是任意数据，然后在其节点下创建一个序列节点，节点名称是 t，数据是任意数据，如果我们查看 task 下的节点，会有以下输出：

```
>ls /task
[t0000000000, t0000000001]
```

> **注意：**创建临时节点参数-e 常常和-s 混用，以实现后面要提到的各种协调功能。

- get，获取节点数据。

```
>get /task
data
cZxid = 0x13
ctime = Fri Mar 24 15:52:09 CST 2017
mZxid = 0x13
mtime = Fri Mar 24 15:52:09 CST 2017
```

```
pZxid = 0x15
cversion = 2
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 4
numChildren = 2
```

get 操作第一行返回的是节点数据，其他行是节点内部使用的数据，说明如下：

- cZxid，节点创建时的 zxid；
- mZxid，节点最新一次更新发生时的 zxid；
- ctime，节点创建时的时间戳；
- mtime，节点最新一次更新发生时的时间戳；
- dataVersion，节点数据的更新次数；
- cversion，其子节点的更新次数；
- aclVersion，节点 ACL（授权信息）的更新次数；
- ephemeralOwner，如果该节点为临时节点，ephemeralOwner 值表示与该节点绑定的会话 ID，如果该节点不是临时节点，ephemeralOwner 值为 0；
- dataLength，节点数据的字节数；
- numChildren，子节点个数。

- delete，删除节点。

```
> delete /task/t0000000000
```

- set path data，设置节点数据。

```
>set /config/db.username xiandafu123
```

- watch 操作，ls 命令和 get 命令都可以增加一个 watch 操作，节点变化的时候会通知客户端。通知完毕后，还需要再次调用 ls 或者 get 才能监听此节点变化。

进入命令行窗口 A，执行以下操作：

```
>get /config/db.username watch
```

然后进入命令行窗口 B，执行以下操作：

```
>set /config/db.username xiandafu
```

这时候，命令行窗口 A 会有以下信息：

```
WATCHER::
WatchedEvent state:SyncConnected type:NodeDataChanged
path:/config/db.username
```

### 16.1.4 领导选取演示

本节将使用命令行工具 zkCli 来演示如何完成分布式协调中的领导选取，在 Spring 集成这一节中，将使用 Curator 来实现这一个过程。

首先创建一个根节点，比如/election，这个节点的主要用途是节点下集合了所有的候选者。每一个候选者（每一个 zk 客户端）都是一个临时的序列节点：

```
>create -s -e /election/e
```

再用命令行工具 zkCli 打开多个窗口，模拟候选者，每个候选者都会调用上面的命令创建一个临时的序列节点。此时查看 election 节点，可能看到以下候选者：

```
>ls /election
[e0000000004, e0000000003, e0000000002, e0000000001, e0000000000]
```

每个候选者使用 zk 创建一个节点的时候，都会知道自己创建的节点名，在选举过程中，用自己的节点名同/election 目录下所有节点进行比较，如果自己的序列号就是最小的序列号，那自己就是当仁不让的领导者节点。

其他节点同时还需要 watch 领导节点，即序列号最小的节点，如果领导节点被删除/退出或者意外宕机，则所有候选者都会收到消息，再次进行如上所述的选举过程。

这种选举过程比较简单，唯一的问题是如果有大量候选者，节点变化同时通知其他大量的候选者再次进行选举会对 zk 有一定的性能影响，因此，一般改进的方式是每个候选者仅监听比自己序列号小的那个候选者。这样，如果领导节点被删除，则序列号较大的候选者能收到领导节点变化事件，只有这一个节点完成选举过程，确认自己是领导节点。

### 16.1.5 分布式锁演示

实现分布式锁可以利用节点唯一性，比如创建一个/locks/xxx 的节点，这里的 xxx 是任意名字，例如对应到业务逻辑的合同号等。

如果创建节点成功，则认为自己获得了锁，可以进行业务操作，如果创建失败，则监听此节点，等待节点被删除。

业务操作完毕后，可以删除此节点。这时候其他客户端将得到 watch 事件，再次创建/locks/xxx，成功则意味着再次获得这个锁。

这种分布式算法较为简单，但同领导选举一样，一旦节点被删除，则会广播监听事件，并且所有候选者都会争相创建节点，性能较差。另外，这种分布式锁算法的问题在于不能随时查看有多少客户端在等待这个锁，以及到底是哪个客户端获取到了这个锁。实际上，也可以用类似领导选举那样的算法来实现分布式锁，谁是领导节点，谁就相当于获得了一个锁。释放锁，将自己创建的节点删除就可以；需要查看有多少客户端在等待锁，只需要查看有多少节点就可以了。

我们将在 Spring Boot 中引用 Curator，这个工具类会帮助我们在不了解 zk 的情况下也能通过简单 API 调用实现领导选举、分布式锁等 zk 提供的协调功能。

### 16.1.6 服务注册演示

我们在 Spring Session 这一章已经看到，通过 Nginx 作为反向代理，可以将客户端请求派发到 Spring Boot 应用上，这种模式对于处理系统之间的调用，比如系统之间通过 REST 调用，会有一些问题：

- Spring Boot 应用的增减还需要修改 Nginx 的配置；
- Nginx 本身也容易出现单点故障。

通过 zk 可以实现服务的注册和服务发现，其算法如下：

对于服务提供者来说，创建一个服务节点，以积分服务为例，比如在/service/credit 目录下创建一个临时序列节点，并设置节点数据为自己服务的 URL。

首先事先创建一个放置服务列表的节点/service/credit：

```
>create /service data
>create /service/credit data
```

提供积分服务的 Spring Boot 应用启动后，需要注册一下自己的服务地址，命令如下：

```
>create -e -s /service/credit/s- 192.168.0.1:8080
Created /service/credit/s-0000000000
```

以上命令创建了一个临时序列节点 s-0000000000，每个提供积分服务的 Spring Boot 应用都在 credit 节点注册自己的服务地址。任何一个调用积分服务的客户端，可以读取/service/credit/的所有节点，随机选择某一个节点，读取节点数据所提供的服务地址，类似以下操作：

```
>ls /service/credit
[s-0000000001, s-0000000000]
>get /service/credit/s-0000000001 watch
192.168.0.1
cZxid = 0x3db4
ctime = Wed Jul 12 09:29:01 CST 2017
...
```

通过服务节点 s-0000000001 的数据就可以知道服务提供的地址是 192.168.0.1。

在获取此节点数据的时候，同时设置了 watch 标记，一旦该节点被删除，比如创建此节点的应用宕机，则客户端能接收到通知，客户端再次从/service/credit 中获取一个可用的服务。

有时候，为了更好地管理服务端和客户端，服务节点可以创建在/service/credit/provider 目录下，而客户端也可以注册到/service/credit/consumer 目录下，consumer 下的节点数据包含了客户端本身的地址和调用的服务端地址。这样有助于整个服务的管理，比如国内的 dubbox 就是采用了这样的服务注册机制。

## 16.2 Spring Boot 集成 ZooKeeper

ZooKeeper 本身提供了低级别的 Java API 来实现前面讲的节点操作。Curator 是 Apache 提供的一个访问 zk 的工具包，封装了这些低级别操作，同时也提供一些高级服务，比如分布式锁、领导选取等，它具有如下特性：

- 自动重连，无须开发人员关心。
- 提供简单的 API 来操作 zk 节点，还有 zk 事件，API 是链式操作风格。
- Curator 实现了 ZooKeeper 提供的所有应用场景（除了两阶段提交），有以下实现
  - 领导节点选取；
  - 分布式锁；

- 分布式读写锁；
- 共享信号量；
- 栅栏和双重 Double Barrier；
- 分布式计数器，支持 integer 和 long；
- 分布式队列和分布式优先级队列；
- 服务注册和发现。

本节主要介绍领导节点选取、分布式锁、服务注册和发现，其他未介绍的实现需要参考 Curator 官网。

## 16.2.1　集成 Curator

集成 Curator，需要向 Spring Boot 的 pom 中添加以下依赖：

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>2.12.0</version>
</dependency>
```

Curator 3.x.x 适用于 ZooKeeper 3.5，并增加了新特性，Curator 2.x.x 适合 ZooKeeper 3.4.x 和 ZooKeeper 3.5.x，截止到本章完稿时，ZooKeeper 3.5 仍然没有正式发布，因此本章使用 2.x 版本。

创建一个 Configuration 类，需要创建一个 CuratorFramework，这是一个线程安全的类，你可以用它完成所有的 zk 功能，代码如下：

```
@Configuration
public class ZookeeperConf {
  @Value("${zk.url}")
  private String zkUrl;

  @Bean
  public CuratorFramework getCuratorFramework(){
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
    CuratorFramework client = CuratorFrameworkFactory.newClient(zkUrl,
retryPolicy);
    client.start();
```

```
        return client;
    }
}
```

RetryPolicy 用于重连策略，当某种原因导致 zk 不可用的时候进行重连尝试的策略，如上例中，ExponentialBackoffRetry 是一种重连策略，每次重连的间隔会越来越长，1000 毫秒是初始化的间隔时间，3 代表尝试重连次数。

CuratorFramework 通过 zkUrl 和 retryPolicy 构造，必须调用 start 开始连接 ZooKeeper。

zkUrl 是 ZooKeeper 的连接地址，来源于 application.properties，内容如下：

```
zk.url=127.0.0.1:2181
```

完成上述集成工作，再次启动 Spring Boot，看到的提示如下，表示集成完成：

```
o.a.c.f.imps.CuratorFrameworkImpl    : Starting
.....
o.a.c.f.imps.CuratorFrameworkImpl    : Default schema
org.apache.zookeeper.ClientCnxn      : Socket connection established,....
org.apache.zookeeper.ClientCnxn      : Session establishment complete on server
```

## 16.2.2 Curator API

Curator API 是链式调用风格，遇到 forPath 接口就触发 ZooKeeper 调用，比如创建一个节点：

```
client.create().forPath("/head", new byte[0]);
client.create().withMode(CreateMode.EPHEMERAL_SEQUENTIAL).forPath("/head
/child", new byte[0]);
```

第一行执行创建命令，在根目录下创建一个 head 节点，设置空字节数据。

第二行执行创建命令，创建一个临时节点，路径是/head/child，设置空字节数据。

- create，创建节点，也可通过 withMode 为节点设置类型，如临时节点、序列节点，以 forPath 结尾。如果节点已经存在，抛出 NodeExistsException。
- delete，删除节点，以 forPath 结尾，如果节点不存在，将抛出 NoNodeException，如果节点是非空节点，则抛出 NotEmptyException。

- checkExists()，检查节点是否存在，如果不存在，返回 null，如果存在，返回 Stat 对象。可以加上 watch 方法来监听节点变化，该方法以 forPath 结尾。

  Stat 对象包含了节点数据，如节点类型、节点 id、版本号、子节点个数、内容长度等，详情参考 16.1.3 节的 get 命令。

- getData，获取节点数据，以 forPath 结尾，返回 byte[]。
- setData，设置节点数据，以 forPath 结尾，比如：

```
zkClient.setData().forPath(path, data.getBytes());
byte[] bs = zkClient.getData().forPath(path);
```

- getChildren()，得到节点的子节点，以 forPath 结尾，返回一个子节点路径列表。

对于 checkExists,getData 和 getChildren，还可以设置 watch 监听方法，比如：

```
zkClient.checkExists().watched().forPath(path);
```

这样，当节点变化时，将通知 Curator，只需要添加一个 CuratorListener 即可，以下为添加一个 CuratorListener：

```
client.getCuratorListenable().addListener(new CuratorListener() {
  public void eventReceived(CuratorFramework client, CuratorEvent event)
throws Exception {
    CuratorEventType type = event.getType();
    if (type == CuratorEventType.WATCHED) {
     WatchedEvent we = event.getWatchedEvent();
     EventType et = we.getType();
     log.info(et + ":" + we.getPath());
     client.checkExists().watched().forPath(we.getPath());
    }
  }

});
```

一般而言，主要监听的就是 WATCHED 事件，这和 zk 的 watch 事件是一样的。

Curator API 支持异步执行，通过在调用链式方法中加入 backgroud()实现。异步的执行结果也将通过 CuratorListener 通知，除了上面提到的 CuratorEventType.WATCHED，还支持后台执行的 CREATE、DELETE、EXISTS 等操作，这里不再一一列举。

> **注意**：zk 得到监听消息后，客户端还必须再设置一次监听，才能收到后面的节点变化事件。
>
> 笔者认为 Curator 通过一个监听器实现两种类型的事件监听，即 zk 的 watch 事件和 Curator 的后台操作事件监听，这种设计模式不好，职责不清楚，应该分别由两种不同的 Listener 来完成。

## 16.3 实现分布式锁

Curator 提供了 zk 场景的绝大部分实现，使用 Curator，就不必关心其内部算法，Curator 提供了 InterProcessMutex 来实现分布式锁，InterProcessMutex 用 acquire 方法获取锁，以及用 release 释放锁，同其他锁一样，release 方法需要放在 finally 代码块中，确保锁能正确释放。

首先创建一个 service 方法：

```
@Service
public class OrderServiceImpl implements OrderService {
  Log log = LogFactory.getLog(OrderServiceImpl.class);
  @Autowired
  CuratorFramework zkClient ;
  String lockPath = "/lock/order";
  // 处理某种订单类型，比如 type 值是"book"
  public void makeOrderType(String type){
  }
}
```

我们需要实现在 makeOrderType 方法上加锁，确保同一个时刻，只能有一个 Spring Boot 应用能执行同此订单类型号相关的操作，其他应用处于等待状态。

```
public void makeOrderType(String type) {
  String path = lockPath+"/"+type;
  log.info("try do job for "+type);
  try{
    InterProcessMutex lock = new InterProcessMutex(zkClient, path);
    if ( lock.acquire(10, TimeUnit.HOURS) ){
      try {
```

```
        // 模拟耗时5秒
        Thread.sleep(1000*5);
        // 即使获得分布式锁，在实际业务处理过程中，也应该检查数据是否已经被处理
        log.info("do job "+type+"done");
      }
      finally{
        lock.release();
      }
    }
  }catch(Exception ex){
    // zk异常
    ex.printStackTrace();
  }
}
```

我们首先确定分布式锁的路径：

```
String path = lockPath+"/"+type;
```

这样，Curator会按照前面所述的算法，在此节点下面创建一系列临时的序列节点，并选择序列最小的节点为锁的拥有者。

```
InterProcessMutex lock = new InterProcessMutex(zkClient, path);
```

InterProcessMutex构造了一个分布式锁，紧接着调用acquire来获取锁操作，并传入一个等待时间，本例中为等待10小时。

如果获得了这个分布式锁，便可以执行业务操作，上面的代码模拟了一个耗时5秒的业务操作。

代码最后，在finally中释放该锁。

如果同时刷新Controller访问此订单处理方法，会看到如下的日志输出：

```
2017-07-11 23:24:36.112  [nio-8080-exec-2] OrderServiceImpl: try do job for
book
2017-07-11 23:24:36.369  [nio-8080-exec-3] OrderServiceImpl: try do job for
book
2017-07-11 23:24:36.495  [nio-8080-exec-4] OrderServiceImpl: try do job for
book
```

```
2017-07-11 23:24:41.117  [nio-8080-exec-2] OrderServiceImpl: do job book done
2017-07-11 23:24:46.125  [nio-8080-exec-3] OrderServiceImpl: do job book done
```

第一个请求（线程名字是 nio-8080-exec-2）在 23:24:36 执行订单处理过程，5 秒后，也就是 23:24:41 执行完，释放锁后，其他 nio-8080-exec-3 和 nio-8080-exec-4 线程都处于等待过程中。

可以在代码块中打上断点，多次刷新 Controller，通过 zk 客户端访问/lock/order/，会看到如下数据：

```
> ls /lock/order/book
[_c_0e2596b9-cdf2-4cc8-80bf-098203767b43-lock-0000000006,
_c_ce97c6cd-af2e-475e-866a-f836c86ab23d-lock-0000000007,
_c_76a61b8c-5f53-4186-a60e-8a96fb3c9ee5-lock-0000000008]
```

book 下有三个节点在竞争锁，其中序列号较大的是 0000000007、0000000008 等待锁，可以通过 get 方法获取到节点数据，里面的 ctime 表示节点创建时间，也就是等待锁的开始时间，节点数据是 IP 地址，可以用这个来判断是哪个客户端在等待此锁。

```
>get /lock/order/book/_c_ce97c6cd-af2e-475e-866a-f836c86ab23d-lock-0000000007
127.0.0.1
cZxid = 0x3da9
ctime = Tue Jul 11 23:35:55 CST 2017
mZxid = 0x3da9
mtime = Tue Jul 11 23:35:55 CST 2017
pZxid = 0x3da9
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x15d322a21c30004
dataLength = 9
numChildren = 0
```

通过获取 0000000007 节点的数据可以判断等待的锁来自客户端“127.0.0.1”，且从 ctime 可以得到视图获取锁的时间。

> **注意：** 正常情况下，再进入锁的时候，应该先判断业务是否已经被处理过了，如果被处理了，则立即退出。

> 如果熟悉 Spring AOP,可以自己完成一个分布式锁注解@ClusterLock 实现，并通过 Curator 来完成分布式锁，比如类似如下：
>
> ```
> @ClusterLock("/lock/order")
> public void makeOrderType(String type){
>
> }
> ```
>
> 这样看起来更酷、更通用。

## 16.4 服务注册

Curator 提供了一个服务注册与发现的封装库，需要在 pom 中添加以下依赖：

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-x-discovery</artifactId>
  <version>2.12.0</version>
</dependency>
```

### 16.4.1 通过 ServiceDiscovery 注册服务

通过 ServiceInstanceBuilder 构造一个服务描述，以及通过 ServiceDiscovery 注册服务：

```
protected void registerSerivce(CuratorFramework client) throws Exception {

  // 构造一个服务描述
  ServiceInstanceBuilder<Map> service = ServiceInstance.builder();
  service.address("192.168.1.100");
  service.port(8080);
  service.name("book");
  Map config = new HashMap();
  config.put("url", "/api/v3/book");
  service.payload(config);

  ServiceInstance<Map> instance = service.build();
```

```
  ServiceDiscovery<Map> serviceDiscovery = ServiceDiscoveryBuilder.builder
(Map.class)
      .client(client).serializer(new JsonInstanceSerializer<Map>(Map.class))
      .basePath("/service").build();
  // 服务注册
  serviceDiscovery.registerService(instance);
  serviceDiscovery.start();

}
```

ServiceInstanceBuilder 是一个描述服务的类，address 方法设置了服务地址，如果没有调用，Curator 会自动设置本机地址，port 参数设置了服务端口，name 参数设置了服务名字，本例中是 name。Curator 会根据这个名字创建一个 zk 节点。ServiceInstanceBuilder 还允许设置 payload，可以是任何类，用来放置额外的信息，本例采用 map，也可以采用自定义的类。

ServiceDiscoveryBuilder 用于创建 ServiceDiscovery 类，用于注册服务，以下方法需要说明：

- client，设置 CuratorFramework。
- serializer，设置序列化类，采用了 Jackson 作为序列化类。
- basePath，指定服务注册的根节点，本例中是/service。

调用上述代码后，可以查看 zk 节点，看到 book 下有一个节点 d4cb62b5-372d-4004-9899-c2e16f83468f，通过 get 查看数据：

```
>ls /service/book
[d4cb62b5-372d-4004-9899-c2e16f83468f]
>get /service/book/d4cb62b5-372d-4004-9899-c2e16f83468f
{"name":"book","id":"d4cb62b5-372d-4004-9899-c2e16f83468f","address":"19
2.168.1.100","port":8080,"sslPort":null,"payload":{"@class":"java.util.HashM
ap","url":"/api/v3/book"},"registrationTimeUTC":1499841184919,"serviceType":
"DYNAMIC","uriSpec":null,"enabled":true}
cZxid = 0x3e00
ctime = Wed Jul 12 14:33:04 CST 2017
```

## 16.4.2 获取服务

获取服务也是调用 ServiceDiscovery 类来实现的，通过调用 queryForInstances 可以获得当前所有可用服务。

```
  protected ServiceInstance<Map> findService(CuratorFramework client,String
serviceName) throws Exception {

    ServiceDiscovery<Map> serviceDiscovery =
ServiceDiscoveryBuilder.builder(Map.class)
      .client(client).serializer(new JsonInstanceSerializer<Map>(Map.class))
      .basePath("/service").build();

    serviceDiscovery.start();
    // 取得当前所有服务
    Collection<ServiceInstance<Map>> all =
serviceDiscovery.queryForInstances(serviceName);
    if(all.size()==0){
      return null;
    }else{
      // 取得第一个服务
      ServiceInstance<Map>  service = new ArrayList<ServiceInstance<Map>>
(all).get(0);
      System.out.println(service.getAddress());
      System.out.println(service.getPayload());
      return service;
    }
  }
```

本例中只简单地取得第一个服务，实际应用中可以随机选择一个服务并保存起来供客户端获取信息，或者采用轮询算法公平地调度服务。

当客户端调用服务后发现其不可用时，可以再次调用这个方法获取一个可用的服务。

## 16.5 领导选取

分布式应用中，有时候必须选出一个节点来分配任务给其他节点，或者负责协调其他节点，比如有多台机器协作处理一批任务，那这些任务应该由哪台机器来分配呢？必须选出一个领导节点来负责。如果领导节点不可用，则在剩下的机器里再选出一个领导节点。

使用 Curator，无须关心 zk 的领导节点的选取算法，通过 LeaderSelector 即可实现领导的选

取，代码如下（这段代码通常在 Curator 初始化配置类中，参考 16.2.1 节）：

```
// 构造一个监听器
LeaderSelectorListenerAdapter listener = new LeaderSelectorListenerAdapter() {
  public void takeLeadership(CuratorFramework client) throws Exception {
    log.info("get leadership");
    // 领导节点，方法结束后退出领导。zk会再次重新选择领导
  }
};

LeaderSelector selector = new LeaderSelector(client, "/schedule", listener);
selector.autoRequeue();
selector.start();
```

# 第 17 章
# 监控Spring Boot应用

Java EE 规范中由 JMX 来监控管理应用，Spring Boot 也提供了 Actuator 功能来完成类似的监控，通过 HTTP、JMX，甚至是远程脚本（SSH）来查看 Spring Boot 应用的配置、各种指标、健康程度等。它能查看和监控以下信息：

- Spring Boot 的配置信息；
- Spring Boot 配置的 Bean 信息；
- 最近请求的 HTTP 信息；
- 数据源，NoSQL 等数据状态；
- 在线查看日志内容，在线日志配置修改；
- 所有@RequestMapping 注解的 URL 路径；
- 自动装配信息汇总；
- 打印虚拟机的线程栈；
- Dump 内存；
- 应用的各种指标汇总；
- 自定义监控指标。

## 17.1 安装 Acutator

使用 Acutator 功能与 Spring Boot 使用其他功能一样简单，只需要在 pom 中增加如下依赖：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
</dependencies>
```

Spring Boot 2 默认并不启用所有的监控，需要做如下配置：

```
endpoints.default.web.enabled=true
```

考虑到系统监控涉及系统安全，因此，最好修改配置文件，设置系统监控的访问端口，如：

```
management.port=8081
```

然后将 8081 置于防火墙后，保证不能非法访问 Spring Boot 的 Acutator 功能。启动 Spring Boot 应用，看到如下信息，表示 Acutator 安装成功：

```
WebMvcEndpointHandlerMapping : Mapped "{[/application/auditevents]...
WebMvcEndpointHandlerMapping : Mapped "{[/application/beans]
WebMvcEndpointHandlerMapping : Mapped "{[/application/health]
WebMvcEndpointHandlerMapping : Mapped "{[/application/status]
WebMvcEndpointHandlerMapping : Mapped "{[/application/loggers
WebMvcEndpointHandlerMapping : Mapped "{[/application/trace]

......
Tomcat started on port(s): 8081 (http)
```

你可以尝试访问：

http://127.0.0.1:8081/application/health

浏览器会有如下输出：

```
{"status":"UP","diskSpace":{"status":"UP","total":120108089344,"free":15
390957568,"threshold":10485760}}
```

health 信息默认输出了磁盘空间的健康诊断信息。

默认情况下，所有的监控都在/application 下，你可以更改配置，修改成你想要的路径：

```
management.context-path=/manage
```

## 17.2 HTTP 跟踪

Spring Boot 提供了 trace 跟踪功能，能查看最近的 HTTP 请求和响应，在浏览器输入：

```
http://localhost:8081/application/trace
```

会输出 HTTP 访问信息：

```
[
  {
      "timestamp": 1499916398894,
      "info": {
          "method": "GET",
          "path": "/all.html",
          "headers": {
              "request": {
                  "host": "127.0.0.1:8080",
                  "connection": "keep-alive",
                  "upgrade-insecure-requests": "1",
                  "user-agent":......,
                  "accept":......,
                  "accept-encoding": "gzip, deflate, sdch, br",
                  "accept-language": "zh-CN,zh;q=0.8",
                  "cookie": ......
              },
              "response": {
                  "Content-Type": "text/html;charset=UTF-8",
                  "Content-Length": "11",
                  "Date": "Thu, 13 Jul 2017 03:26:38 GMT",
                  "status": "200"
              }
          },
          "timeTaken": "3"
      }
  }
]
```

info 对象包含了 path 属性和 method，path 是获取/all.html，headers 包含了请求 HTTP 头内容，可以帮助我们诊断一些系统问题。

trace 是通过 InMemoryTraceRepository 类来实现的，默认保留最后 100 条访问数据，你可以自己配置 InMemoryTraceRepository 或者实现 TraceRepository 接口，以下为默认保留最后访问的 2 条数据：

```
@Configuration
public class ActuatorConfig {
  @ConditionalOnMissingBean(TraceRepository.class)
  @Bean
  public InMemoryTraceRepository traceRepository() {
    InMemoryTraceRepository httpTrace = new  InMemoryTraceRepository();
    httpTrace.setCapacity(2);
    return httpTrace;
  }
}
```

## 17.3 日志查看

Actuator 允许查看日志配置，还允许修改日志等级配置，Actuator 也可以在线查看日志内容。在浏览器中输入：

```
http://localhost:8081/application/loggers
```

会有以下输出（截取部分片段）：

```
{"levels":["OFF","ERROR","WARN","INFO","DEBUG","TRACE"],"loggers":{"ROOT":
{"configuredLevel":"INFO","effectiveLevel":"INFO"},"com":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee.sample":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee.sample.ch17":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee.sample.ch17.Ch17Ap
plication":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee.sample.ch17.
controller":
{"configuredLevel":null,"effectiveLevel":"INFO"},"com.bee.sample.ch17.co
ntroller.TestCrontroller":{"configuredLevel":null,"effectiveLevel":"INFO"},
```

```
......
}}}
```

可以看到，TestController 类的日志等级是 INFO（configuredLevel 是指配置的日志等级，本例中并未配置日志等级，因此默认的是 INFO）。

可以通过提交以下 POST 片段来动态改变日志等级，比如将 TestController 的日志等级改为 DEBUG：

```
>curl -XPOST
'http://localhost:8081/application/loggers/com.bee.sample.ch17.controller'
-H 'Content-Type: application/json' -d'
  {
    "configuredLevel": "DEBUG"
  }
'
```

可以通过 application/logfile 来查看日志，如果配置日志文件为 my.log：

```
logging.file = my.log
```

通过浏览器可以查看日志文件：

```
http://localhost:8081/application/logfile
```

如果想查看指定范围的日志，则需要在 HTTP 头增加 Range 参数：

```
curl -XGET 'http://localhost:8081/application/logfile' -H 'Range:bytes=0-499'
```

Range 头来自于 HTTP 协议，表示获取资源的内容片段（这正是 HTTP 断点续传的基础），上例中获取了日志文件头 500 个字节的内容。

通常，查看日志总是从文件末尾向前查看，因此，可以使用 Range:bytes=-499 来查看日志文件最后的 499 个字节。

> logfile 目前的功能还不具备实用功能，通过 Range 头来查看范围并不适合在浏览器里输入，因此最好的方式应该支持类似以下的查看，比如查看最后 500 行：
>
> ```
> http://localhost:8081/logfile?range=-500
> ```
>
> 遗憾的是目前 Spring Boot 并没有支持这种用法。

## 17.4 线程栈信息

在 Spring Boot 中，可以通过输入 dump 来获取某一时刻虚拟机线程栈信息，该信息类似使用 JDK 自带的 jstack 命令的输出结果或者 kill -3 的结果。线程栈表示某一时刻虚拟机正在做的事情，比如以下输出：

```
http://127.0.0.1:8081/application/dump
```

会得到如下效果：


```
......
{
  "threadName":"http-nio-8080-exec-1",
  "threadId":574,
  "blockedTime":-1,
  "blockedCount":0,
  "waitedTime":-1,
  "waitedCount":2,
  "lockName":null,
  "lockOwnerId":-1,
  "lockOwnerName":null,
  "inNative":false,
  "suspended":false,
  "threadState":"TIMED_WAITING",
  "stackTrace":[
  {
  "methodName":"sleep",
  "fileName":"Thread.java",
  "lineNumber":-2,
  "className":"java.lang.Thread",
  "nativeMethod":true
  },
  {
  "methodName":"sleepTest",
  "fileName":"TestCrontroller.java",
  "lineNumber":19,
  "className":"com.bee.sample.ch19.controller.TestCrontroller",
  "nativeMethod":false
  },
```

```
    ......
    {
"methodName":"invokeForRequest",
    "fileName":"InvocableHandlerMethod.java",
    "lineNumber":133,

"className":"org.springframework.web.method.support.InvocableHandlerMethod",
    "nativeMethod":false
    },
    ......
}
```

以上截取了线程栈某一个线程的运行情况，线程名字是 http-nio-8080-exec-1，以 http-nio-8080-exec 开头的线程总是用来处理浏览器端发起的 HTTP 请求，因此我们可以观察到此时发生了什么。如上所示，从 threadState 的值观察，这个线程处于 TIMED_WAITING 状态，也就是线程处于等待状态，stackTrace 是线程调用栈，可以看到代码所在类是 TestController，发生在 19 行，调用了 Thread.sleep()方法。

这个线程栈很好地解释了发起的 http://localhost:8080/testsleep.html 为什么一直没有响应。如果查看代码，会看到如下内容：

```
@RequestMapping("/testsleep.html")
public @ResponseBody String sleepTest() throws Exception{
  Thread.currentThread().sleep(1000*1000);
  return "";
}
```

这个代码有点愚蠢，但现实情况是 Spring Boot 的应用较为复杂，从庞大的代码库中找到问题并不是那么简单，查看线程栈是个很好的办法，通过观察 dump 打印出来的线程栈，还能轻易发现应用死锁、系统莫名忙碌等系统故障的原因。

以系统莫名的 CPU 使用率高来说，你可以打开两个浏览器，在相隔一秒的时候，分别打印出 dump 信息，然后观察 http-nio-8080-exec 开头的线程栈，如果某一个线程栈在一秒时间内都停留在同一个业务代码位置，则表示系统在那里可能出问题了。你可以精确地锁定系统忙的原因及所在代码行。

如果系统响应慢，也可以像上面一样观察线程栈，观察到系统正在干什么，笔者有一次去帮助看一个项目为什么上传文件很很慢，后来通过隔 2 秒打印几个线程栈，发现所有线程栈资源都处于正常运行、等待接受服务状态。断定并非 Spring Boot 的问题，进而通过 17.2 节提到的 trace 功能，发现根本没有发起任何上传请求。最后断定问题是由于上传的 activex 控件出现

了问题导致的。

> 使用线程栈来诊断系统问题，远远超出了本书的知识范畴，建议访问 Java 官网，了解 Java 的 jstack 命令。

## 17.5 内存信息

Spring Boot 的 heapdump 类似 JDK 提供的 jmap 工具，能将内存镜像压缩下载以提供分析，在浏览器中输入：

```
http://localhost:8081/application/heapdump
```

内存镜像将以 gzip 的格式压缩后下载。

> 内存镜像通常是查看内存溢出最好的办法，获得内存镜像更通用的方法是使用 JDK 提供的 jmap 命令，还可以在 Spring Boot 应用启动的时候增加如下参数：
>
> ```
> XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/xxx/file.hprof
> ```
>
> 这样，系统内存溢出的时候自动将当时的内存情况镜像到你指定的文件中。

分析内存镜像有很多工具，JDK 提供了 jhat 来读取内存镜像文件，并通过 OQL 对象查询语言来分析内存。这个同线程栈分析一样，已经超出了本书的范畴，这里简单地举一个例子，假定如下调用 putdata.html 多次后，会出现内存溢出情况（实际需要调用非常多次，但每次调用都会占用一定内存）：

```
@Controller
public class OutofMemoryController {
  List<String> list = Collections.synchronizedList(new ArrayList<String>());

  @RequestMapping("/putdata.html")
  public @ResponseBody String create(){
    addData();
      return "success "+list.size();
  }

  private void addData(){
    // 每次放入 300 条数据
    for(int i=0;i<300;i++){
```

```
            String s =  "abcd"+System.currentTimeMillis();
            list.add(s);
        }
    }
}
```

我们可以在 Spring Boot 应用中加入上面提到的虚拟机参数，在内存溢出后分析镜像文件，或者直接使用 Actuator 的 heapdump 在系统宕机之前分析问题，在浏览器中输入 http://localhost:8081/application/heapdump 后，会下载一个 heapdumpyyyy-MM-dd-live（加时间戳）.hprof.gz，如笔者的 heapdump2017-07-14-00-27-live3358357938795829131.hprof.gz。解压该文件，得到的解压文件是 heapdump2017-07-14-00-27-live3358357938795829131.hprof，这是一个内存镜像文件，需要用 jhat 打开这个文件：

```
localhost:Downloads xiandafu$ jhat -port 7000 heapdump2017-07-14-00-27-live3358357938795829131.hprof
Reading from heapdump2017-05-08-12-54-live3933850398915553412.hprof...
Dump file created Mon May 08 12:54:52 CST 2017
Snapshot read, resolving...
Resolving 387961 objects...
Chasing references, expect 77 dots.............................................................................
Eliminating duplicate references...........................................................................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

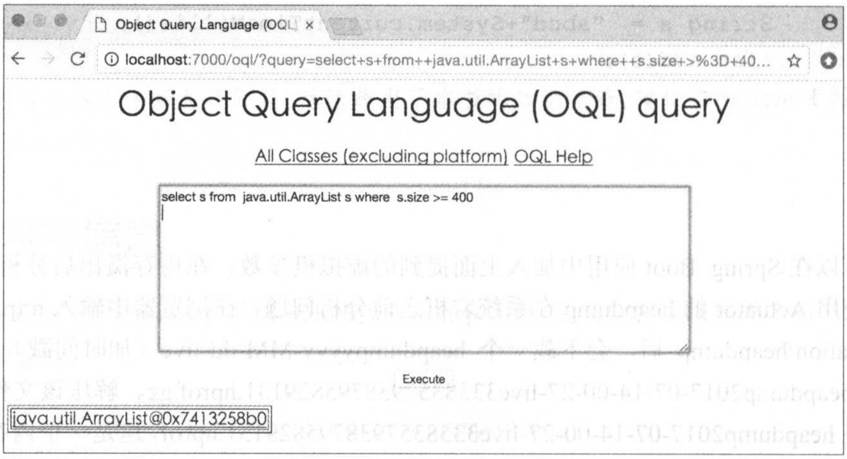
参数-port 指示了 jhat 的访问地址，我们可以通过浏览器访问该地址，进入内存分析界面。

```
http://localhost:7000/
```

首页包含了虚拟机的所有类和实例个数，点击每个实例后，能看到实例当前的属性，这个对于分析内存没什么用，我们需要使用 OQL 对象查询语言来直接查询我们关心的数据，因此在浏览器输入（或者首页底部有 Execute Object Query Language (OQL) query 链接）：

```
http://localhost:7000/oql/
```

会看到如下界面：




OQL 语言类似 SQL，可以通过 select from 来查询和统计根据条件得出来的对象实例。对于内存溢出，我们可以假设某个 List 放入过多数据（也可以假设某个 Map 放入过多数据），因此我们可以通过查询 List 的元素个数大于某个很大的整数来进行判断。我们可以输入以下 OQL 并执行以查看结果：

```
select s from  java.util.ArrayList s where  s.size >= 400
```

以上 OQL 查询了 ArrayList 的所有实例，并筛选出 size 超过 400 的实例。如上图所示，仅仅找到一个实例 **java.util.ArrayList@0x7413258b0**，@后面的数字是实例的 ID，我们点击进入后会看到关于此实例的基本信息等，我们可以点击“References to this object:”，看看到底是什么类引用了此实例。我们不但希望找到内存溢出的原因，也希望找到是谁引起的，我们会看到以下片段：

```
    References to this object:
    com.bee.sample.ch19.controller.OutofMemoryController@0x741325888 (24
bytes) : field list
    java.util.Collections$SynchronizedRandomAccessList@0x741325898 (40
bytes) : field mutex
```

也就是 id 为 0x741325888，类型是 OutofMemoryController 的实例引用了这个 ArrayList，因此，我们可以结合源码，再来分析为何这样会导致内存溢出。

> 关于镜像分析和 OQL 就只介绍到这里，可以参考 https://visualvm.java.net/oqlhelp.html 来了解如何分析内存，同时也可以参考 jmap 和 jhat 命令。Eclipse Memory Analyzer 也可以方便地分析内存故障。

## 17.6 查看 URL 映射

Actuator 的 mappings 输出所有通过注解@RequestMapping 设置的 URL 映射，可以通过此来查看 URL 对应的 Controller：

```
{
    "/webjars/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**/favicon.ico": {
        "bean": "faviconHandlerMapping"
    },
    "{[/all.html]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String
com.bee.sample.ch17.controller.OutofMemoryController.create()"
    },
    "{[/test.html]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String
com.bee.sample.ch17.controller.TestController.create() throws
java.lang.Exception"
    }
}
```

## 17.7 查看 Spring 容器管理的 Bean

Actuator 的 beans 输出所有 Spring 管理的 Bean，输出如下：

```
[
    {
        "context": "application",
        "parent": null,
        "beans": [
            {
```

```
            "bean": "ch17Application",
            "aliases": [ ],
            "scope": "singleton",
            "type":
"com.bee.sample.ch17.Ch17Application$$EnhancerBySpringCGLIB$$b39a6375",
            "resource": "null",
            "dependencies": [ ]
        },
        {
            "bean":
"org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory",
            "aliases": [ ],
            "scope": "singleton",
            "type":
"org.springframework.core.type.classreading.CachingMetadataReaderFactory",
            "resource": "null",
            "dependencies": [ ]
        }
        .....
      ]
  }
]
```

## 17.8 其他监控

- health：查看所在应用的健康状态，如磁盘、数据源、Redis、Elasticsearch 等，将在下一节详细介绍。
- metrics：显示 Spring Boot 的性能指标，如已有内存、未占用内存、垃圾回收次数、类信息等。
- env：显示 Spring Boot 环境变量，如使用的 JDK 版本、加载的 jar 包、配置文件信息、日志文件信息。
- configprops：所有@ConfigurationProperties 注解的配置信息，如文件上传的最大允许配置等。
- autoconfig：显示所有自动装配类的报告，以及是什么原因导致自动装配成功或者不成功。

## 17.9 编写自己的监控信息

health 用于检查 Spring Boot 应用的健康状态，提供了磁盘的健康状态显示，如果应用还使用了数据源、NoSQL 等，也会显示相应的健康状态。Spring Boot 使用 HealthIndicator 接口实现监控信息显示，默认有如下类实现 HealthIndicator 接口。

| 实 现 类 | 描 述 |
|---|---|
| DiskSpaceHealthIndicator | 检查磁盘空间 |
| DataSourceHealthIndicator | 检查数据源是否可用，数据源对应的数据库版本信息 |
| XXXHealthIndicator | 其中 XXX 代表了内置的 Elasticsearch、JMS、Mail、MongoDB、Rabbit、Redis、Solr 等，以验证这些服务是否可用 |

metrics 用来查看系统的各项指标，包括主机内存的大小、虚拟机 heap 信息、线程信息、垃圾回收、Tomcat 会话信息等。也可以在你的 Bean 中注入 CounterService，调用 increment 方法和 decrement 方法来设置指标值，将在 17.9.2 节详细描述。

### 17.9.1 编写 HealthIndicator

编写自己的监控器，只需要继承 HealthIndicator，实现 health 方法，返回一个 Health 对象即可：

```
@Component
public class MessageCenterHealthIndicator implements HealthIndicator {

    public MessageCenterHealthIndicator(){
    }

    @Override
    public Health health() {

        int errorCode = check(); //
        if (errorCode != 0) {
            return Health.down().withDetail("Message", "error:
"+errorCode).build();
        }
        return Health.up().build();
    }
```

```
    protected int check(){
        // 模拟返回一个错误状态
        return 1;
    }
}
```

Health 对象的 up 方法表示健康，对象正常，down 表示异常，可以通过 withDetail 添加任意多的信息，对象的名称默认是类名字去掉 HealthIndicator 后缀。访问 Health，会看到以下信息：

```
{
    "status": "DOWN",
    "messageCenter": {
        "status": "DOWN",
        "Message": "error 1"
    },
    "diskSpace": {
        "status": "UP",
        "total": 120108089344,
        "free": 24940900352,
        "threshold": 10485760
    },
    "db": {
        "status": "UP",
        "database": "MySQL",
        "hello": 1
    }
}
```

### 17.9.2 自定义监控

Spring Boot 2 提供注解@Endpoint 来自定义一个监控类，并在方法上使用@ReadOperation 来显示监控指标，使用@WriteOperation 来动态更改监控指标。

以下是一个数据源为 HikariCP 的监控类，提供了数据源配置的最大连接数、空闲连接数、占用连接数等信息，同时也允许动态设置最大连接数。

```
@Endpoint(id = "hikariCP")
@Bean
public class HikariCPEndpoint {
```

```
    HikariDataSource ds;
    public HikariCPEndpoint(HikariDataSource ds) {
      this.ds = ds;
    }
    @ReadOperation
    public Map<String, Object> info() {
      // 返回监控信息，现在暂时什么都返回
      return new HashMap();
    }

  }
```

HikariCPEndpoint 必须使用注解@Endpoint，这里设置 id 为 hikariCP。然后可以采用如下代码片段将此监控类配置到 Spring Boot 监控中：

```
@Configuration
public class AcutatorExtConfig {
    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnEnabledEndpoint
    public HikariCPEndpoint hikariCPEndpoint(DataSource ds) {
        return new HikariCPEndpoint((HikariDataSource)ds);
    }
}
```

监控类 Endpoint 必须使用@Bean 声明为 Spring 管理 Bean。注解@ConditionalOnMissingBean 表示如果还未声明，HikariCPEndpoint 才会生效，生效的另外一个条件是系统启用了监控，即 17.1 节的启用监控配置。

重新启动 Spring Boot 2 应用，会发现注册了一个名字 hikariCP 的监控：

```
...WebMvcEndpointHandlerMapping : Mapped "{[/application/hikariCP]
...WebMvcEndpointHandlerMapping : Mapped "{[/application/auditevents]
......
```

通过浏览器访问：

http://127.0.0.1:8081/application/hikariCP

浏览器会有如下输出：

```
{}
```

为了提供具体的数据源监控信息，需要@ReadOperation 注解的方法，HikariDataSource 正好提供了多个监控指标，代码实现如下：

```
@ReadOperation
public Map<String, Object> info() {
  HashMap map  = new HashMap();
 // 连接池配置
  HikariConfigMXBean configBean = ds.getHikariConfigMXBean();
  map.put("total",configBean.getMaximumPoolSize());
 // 连接池运行状态
  HikariPoolMXBean mxBean = ds.getHikariPoolMXBean();
  map.put("active", mxBean.getActiveConnections());
  map.put("idle", mxBean.getIdleConnections());
  // 连接都被使用情况下，等待获取连接的线程个数
  map.put("threadWait", mxBean.getThreadsAwaitingConnection());
  return map;
}
```

再次访问/application/hikariCP，Spring Boot 会将返回的 Map 序列化成 JSON 输出到前端，类似如下代码：

```
{"threadWait":0,"total":5,"idle":5,"active":0}
```

使用注解@WriteOperation 可以以 HTTP POST 方式来动态更改监控指标。如下代码会动态地调整数据连接池的最大连接数：

```
@WriteOperation
public void setMax(int max) {
  ds.getHikariConfigMXBean().setMaximumPoolSize(max);
}
```

使用 curl 发起 POST 请求来更改连接数：

```
curl -XPOST 'http://127.0.0.1:8081/application/hikariCP' -H 'Content-Type:
application/json' -d'
{
"max":7
}
```

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

# Spring Boot 2 精髓

## 从构建小系统到架构分布式大系统

Spring Boot是目前Spring技术体系中炙手可热的框架之一，既可用于构建业务复杂的企业应用系统，也可以开发高性能和高吞吐量的互联网应用。Spring Boot框架降低了Spring技术体系的使用门槛，简化了Spring应用的搭建和开发过程，提供了流行的第三方开源技术的自动集成。

本书系统介绍了Spring Boot 2的主要技术，侧重于两个方面，一方面是极速开发一个Web应用系统，详细介绍Spring Boot框架、Spring MVC、视图技术、数据库访问技术，并且介绍多环境部署、自动装配、单元测试等高级特性；另一方面，当系统模块增加，性能和吞吐量要求增加时，如何平滑地用Spring Boot实现分布式架构，也会在本书后半部分介绍，包括使用Spring实现REST-ful架构，在Spring Boot框架下使用Redis、MongoDB、ZooKeeper、Elasticsearch等流行技术，使用Spring Session实现系统水平扩展，使用Spring Cache提高系统性能。

阅读本书的人，可以是Java新手，从未使用过任何Spring技术的工程师。也可以是用过Spring，但想进一步了解Spring Boot的开发者。如果你已经使用过Spring Boot，那么本书也非常适合你全面深入了解Spring Boot。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

上架建议：计算机>微服务

ISBN 978-7-121-32825-1
9 787121 328251 >
定价：79.00元

PHEI
责任编辑：陈晓猛
封面设计：李　玲